SIAM J. COMPUT.
Vol. 25, No. 1, pp. 1–51, February 1996

ASYMPTOTIC CONDITIONAL PROBABILITIES: THE UNARY CASE*

ADAM J. GROVE[†], JOSEPH Y. HALPERN[‡], AND DAPHNE KOLLER[§]

Abstract. Motivated by problems that arise in computing *degrees of belief*, we consider the problem of computing asymptotic conditional probabilities for first-order sentences. Given first-order sentences φ and θ, we consider the structures with domain $\{1, \ldots, N\}$ that satisfy θ, and compute the fraction of them in which φ is true. We then consider what happens to this fraction as N gets large. This extends the work on 0-1 laws that considers the limiting probability of first-order sentences, by considering asymptotic *conditional* probabilities. As shown by Līogon'kīĭ [*Math. Notes Acad. USSR*, 6 (1969), pp. 856–861] and by Grove, Halpern, and Koller [*Res. Rep.* RJ 9564, IBM Almaden Research Center, San Jose, CA, 1993], in the general case, asymptotic conditional probabilities do not always exist, and most questions relating to this issue are highly undecidable. These results, however, all depend on the assumption that θ can use a nonunary predicate symbol. Līogon'kīĭ [*Math. Notes Acad. USSR*, 6 (1969), pp. 856–861] shows that if we condition on formulas θ involving unary predicate symbols only (but no equality or constant symbols), then the asymptotic conditional probability does exist and can be effectively computed. This is the case even if we place no corresponding restrictions on φ. We extend this result here to the case where θ involves equality and constants. We show that the complexity of computing the limit depends on various factors, such as the depth of quantifier nesting, or whether the vocabulary is finite or infinite. We completely characterize the complexity of the problem in the different cases, and show related results for the associated approximation problem.

Key words. asymptotic probability, 0-1 law, finite model theory, degree of belief, labeled structures, principle of indifference, complexity

AMS subject classifications. 03B48, 60C05, 68Q25

1. Introduction.

Suppose we have a sentence θ expressing facts that are known to be true, and another sentence φ whose truth is uncertain. Our knowledge θ is often insufficient to determine the truth of φ: both φ and its negation may be consistent with θ. In such cases, it can be useful to assign a *probability* to φ given θ. In a companion paper [23], we described our motivation for investigating this idea, and presented our general approach. We repeat some of this material below, to provide the setting for the results of this paper.

One important application of assigning probabilities to sentences—indeed, the one that has provided most of our motivation—is in the domain of decision theory and artificial intelligence. Consider an agent (or expert system) whose knowledge consists of some facts θ, who would like to assign a *degree of belief* to a particular statement φ. For example, a doctor may want to assign a degree of belief to the hypothesis that a patient has a particular illness, based on the symptoms exhibited by the patient together with general information about symptoms and diseases. Since the actions the agent takes may depend crucially on this value, we would like techniques for computing degrees of belief in a principled manner.

The difficulty of defining a principled technique for computing the probability of φ given θ, and then actually computing that probability, depends in part on the language and logic being considered. In decision theory, applications often demand the ability to express statistical knowledge (for instance, correlations between symptoms and diseases) as well as first-order

*Received by the editors October 13, 1993; accepted for publication (in revised form) July 7, 1994. Some of this research was performed while Adam Grove and Daphne Koller were at Stanford University and at the IBM Almaden Research Center. A preliminary version of this paper appeared in *Proc. 24th ACM Symp. on Theory of Computing* [20]. This research was sponsored in part by an IBM Graduate Fellowship to Adam Grove, by a University of California President's Postdoctoral Fellowship to Daphne Koller, and by Air Force Office of Scientific Research contract F49620-91-C-0080. The United States Government is authorized to reproduce and distribute reprints for governmental purposes.

[†] NEC Research Institute, 4 Independence Way, Princeton, NJ 08540 (grove@research.nj.nec.com).

[‡] IBM Almaden Research Center, 650 Harry Rd., San Jose, CA 95120 (halpern@almaden.ibm.com).

[§] Computer Science Division, University of California at Berkeley, Berkeley, CA 94720 (daphne@cs.berkeley.edu).

knowledge. Work in the field of 0-1 *laws* (which, as discussed below, is closely related to our own) has examined some higher-order logics as well as first-order logic. Nevertheless, the pure first-order case is still difficult, and is important because it provides a foundation for all extensions. In this paper and in [23] we address the problem of computing conditional probabilities in the first-order case. In a related paper [22], we consider the case of a first-order logic augmented with statistical knowledge.

The general problem of assigning probabilities to first-order sentences has been well studied (cf. [15] and [16]). In this paper, we investigate two specific formalisms for computing probabilities, based on the same basic approach. Our approach is itself an instance of a much older idea, known as *the principle of insufficient reason* [28] or *the principle of indifference* [26]. This states that all possibilities should be given equal probability, and was once regarded as one of the most basic principles of probability theory. (See [24] for a discussion of the history of the principle.) We use this idea to assign equal degrees of belief to all basic "situations" consistent with the known facts. The two formalisms we consider differ only in how they interpret "situation." We discuss this in more detail below.

In many applications, including the one of most interest to us, it makes sense to consider finite domains only. In fact, the case of most interest to us is the behavior of the formulas φ and θ over large finite domains. Similar questions also arise in the area of 0-1 *laws*. Our approach essentially generalizes the methods used in the work on 0-1 laws for first-order logic to the case of conditional probabilities. (See Compton's overview [8] for an introduction to this work.) Assume, without loss of generality, that the domain is $\{1, \ldots, N\}$ for some natural number N. As we said above, we consider two notions of "situation." In the *random-worlds method*, the possible situations are all the worlds, or first-order models, with domain $\{1, \ldots, N\}$ that satisfy the constraints θ. Based on the principle of indifference, we assume that all worlds are equally likely. To assign a probability to φ, we therefore simply compute the fraction of them in which the sentence φ is true. The random-worlds approach views each individual in $\{1, \ldots, N\}$ as having a distinct name (even though the name may not correspond to any constant in the vocabulary). Thus, two worlds that are isomorphic with respect to the symbols in the vocabulary are still treated as distinct situations. In some cases, however, we may believe that all relevant distinctions are captured by our vocabulary, and that isomorphic worlds are not truly distinct. The *random-structures method* attempts to capture this intuition by considering a situation to be a *structure*—an *isomorphism class* of worlds. This corresponds to assuming that individuals are distinguishable only if they differ with respect to properties definable by the language. As before, we assign a probability to φ by computing the fraction of the structures that satisfy φ among those structures that satisfy θ.[1]

Since we are computing probabilities over finite models, we have assumed that the domain is $\{1, \ldots, N\}$ for some N. However, we often do not know the precise domain size N. In many cases, we know only that N is large. We therefore estimate the probability of φ given θ by the asymptotic limit, as N grows to infinity, of this probability over models of size N.

Precisely the same definitions of asymptotic probability are used in the context of 0-1 laws for first-order logic, but without allowing prior information θ. The original 0-1 law, proved independently by Glebskiĭ et al. [18] and Fagin [13], states that the asymptotic probability of any first-order sentence φ with no constant or function symbols is either 0 or 1. This means that such a sentence is true in almost all finite structures, or in almost none.

[1] The random-worlds method considers what has been called in the literature *labeled* structures, while the random-structures method considers *unlabeled* structures [8]. We choose to use our own terminology for the random-worlds and random-structures methods, rather than the terminology of labeled and unlabeled. This is partly because we feel it is more descriptive, and partly because there are other variants of the approach that are useful for our intended application, and that do not fit into the standard labeled/unlabeled structures dichotomy (see [2]).

Our work differs from the original work on 0-1 laws in two respects. The first is relatively minor: we need to allow the use of constant symbols in φ, as they are necessary when discussing individuals (such as patients). Although this is a minor change, it is worth observing that it has a significant impact. It is easy to see that once we allow constant symbols, the asymptotic probability of a sentence φ is no longer either 0 or 1; for example, the asymptotic probability of $P(c)$ is $\frac{1}{2}$. Moreover, once we allow constant symbols, the asymptotic probability under random worlds and under random structures need not be the same. The more significant difference, however, is that we are interested in the asymptotic *conditional* probability of φ, given some prior knowledge θ. That is, we want the probability of φ over the class of finite structures defined by θ.

Some work has already been done on aspects of this question. Līogon'kiĭ [31], and independently Fagin [13], showed that asymptotic conditional probabilities do not necessarily converge to any limit. Subsequently, 0-1 laws were proved for special classes of first-order structures (such as graphs, tournaments, partial orders, etc.; see the overview paper [8] for details and further references). In many cases, the classes considered could be defined in terms of first-order constraints. Thus, these results can be viewed as special cases of the problem that we are interested in: computing asymptotic *conditional* probabilities relative to structures satisfying the constraints of a database. Lynch [32] showed that, for the random-worlds method, asymptotic probabilities exist for first-order sentences involving unary functions, although there is no 0-1 law. (Recall that the original 0-1 result is specifically for first-order logic *without* function symbols.) This can also be viewed as a special case of an asymptotic conditional probability for first-order logic without functions, since we can replace the unary functions by binary predicates, and condition on the fact that they are functions.

The most comprehensive work on this problem is the work of Līogon'kiĭ [31].[2] In addition to pointing out that asymptotic conditional probabilities do not exist in general, he shows that it is undecidable whether such a probability exists. (We generalize Līogon'kiĭ's results for this case in [23].) He then investigates the special case of conditioning on formulas involving unary predicates only (but no constants or equality). In this case, he proves that the asymptotic conditional probability does exist and can be effectively computed, even if the left side of the conditional, φ, has predicates of arbitrary arity and equality. This gap between unary predicates and binary predicates is somewhat reminiscent of the fact that first-order logic over a vocabulary with only unary predicates (and constant symbols) is decidable, while if we allow even a single binary predicate symbol, then it becomes undecidable [11], [29]. This similarity is not coincidental; some of the techniques used to show that first-order logic over a vocabulary with unary predicate symbols is decidable are used by us to show that asymptotic conditional probabilities exist.

In this paper, we extend the results of Līogon'kiĭ [31] for the unary case. We first prove (in §3) that, if we condition on a formula involving only unary predicates, constants, and equality that is satisfiable in arbitrarily large models, the asymptotic conditional probability exists. We also present an algorithm for computing this limit. The main idea we use is the following: to compute the asymptotic conditional probability of φ given θ, we examine the behavior of φ in finite models of θ. It turns out that we can partition the models of θ into a finite collection of classes, such that φ behaves *uniformly* in any individual class. By this we mean that almost all worlds in the class satisfy φ or almost none do; i.e., there is a 0-1 law for the asymptotic probability of φ when we restrict attention to models in a single class. In §3 we define these classes and prove the existence of a 0-1 law within each class. We also

[2] In an earlier version of this paper [21], we stated that, to our knowledge, no work had been done on the general problem of asymptotic conditional probabilities. We thank Moshe Vardi for pointing out to us the work of Līogon'kiĭ [31].

TABLE 1
Complexity of asymptotic conditional probabilities.

	Depth ≤ 1	**Restricted**	**General case**
Existence	NP-complete	NEXPTIME-complete	NEXPTIME-complete
Compute	#P/PSPACE	#EXP-complete	#TA(EXP,LIN)-complete
Approximate	(co-)NP-hard	(co-)NEXPTIME-hard	TA(EXP,LIN)-hard

show how the asymptotic conditional probability of φ given θ can be computed using these 0-1 probabilities.

In §4 we turn our attention to the complexity of computing the asymptotic probability in this case. Our results, which are the same for random worlds and random structures, depend on several factors: whether the vocabulary is finite or infinite, whether there is a bound on the depth of quantifier nesting, whether equality is used in θ, whether nonunary predicates are used, and whether there is a bound on predicate arities. For a fixed and finite vocabulary, there are just two cases: if there is no bound on the depth of quantifier nesting then computing asymptotic conditional probabilities is PSPACE-complete, otherwise the computation can be done in linear time. The case in which the vocabulary is not fixed (which is the case more typically considered in complexity theory) is more complex. The results for this case are summarized in Table 1. Perhaps the most interesting aspect of this table is the factors that cause the difference in complexity between #EXP and #TA(EXP,LIN) (where #TA(EXP,LIN) is the counting class corresponding to alternating Turing machines that take exponential time and make only a linear number of alternations; a formal definition is provided in §4.5). If we allow the use of equality in θ, then we need to restrict both φ and θ to using only unary predicates to get the #EXP upper bound. On the other hand, if θ does not mention equality, then the #EXP upper bound is attained as long as there is some fixed bound on the arity of the predicates appearing in φ. If we have no bound on the arity of the predicates that appear in φ, or if we allow equality in θ and predicates of arity 2 in φ, then the #EXP upper bound no longer holds, and we move to #TA(EXP,LIN).

Our results showing that computing the asymptotic probability is hard can be extended to show that finding a nontrivial estimate of the probability (i.e., deciding if it lies in a nontrivial interval) is almost as difficult. The lower bounds for the arity-bounded case and the general case require formulas of quantification depth 2 or more. For unquantified sentences or depth-1 quantification, things seem to become an exponential factor easier. We do not have tight bounds for the complexity of computing the degree of belief in this case; we have a #P lower bound and a PSPACE upper bound. The results for depth 1 are not proved in this paper; see [27] for details.

We observe that apart from our precise classification of the complexity of these problems, our analysis provides an effective algorithm for computing the asymptotic conditional probability. The complexity of this algorithm is, in general, double-exponential in the number of unary predicates used and in the maximum arity of any predicate symbol used; it is exponential in the overall size of the vocabulary and in the lengths of φ and θ.

Our results are of more than purely technical interest. The random-worlds method is of considerable theoretical and practical importance. We have already mentioned its relevance to computing degrees of belief. There are well-known results from physics that show the close connection between the random-worlds method and *maximum entropy* [25]. These results say that in certain cases the asymptotic probability can be computed using maximum entropy methods. Some formalization of similar results, but in a framework that is close to that of the current paper, can be found in [33] and [22]. (These results are of far more interest when there are statistical assertions in the language, so we do not discuss them here.)

As we observe in [23] and [22], this connection relies on the fact that we are conditioning on a unary formula. In fact, in almost all applications where maximum entropy has been used (and where its application can be best justified in terms of the random-worlds method) the knowledge base is described in terms of unary predicates (or, equivalently, unary functions with a finite range). For example, in physics applications we are interested in such predicates as quantum state (see [10]). Similarly, AI applications and expert systems typically use only unary predicates [7] such as symptoms and diseases. In general, many properties of interest can be expressed using unary predicates, since they express properties of individuals. Indeed, a good case can be made that statisticians tend to reformulate all problems in terms of unary predicates, since an event in a sample space can be identified with a unary predicate [36]. Indeed, in most cases where statistics are used, we have a basic unit in mind (an individual, a family, a household, etc.), and the properties (predicates) we consider are typically relative to a single unit (i.e., unary predicates). Thus, results concerning computing the asymptotic conditional probability if we condition on unary formulas are significant in practice.

2. Definitions. Let Φ be a set of predicate and function symbols, and let $\mathcal{L}(\Phi)$ (resp., $\mathcal{L}^-(\Phi)$) denote the set of first-order sentences over Φ with equality (resp., without equality). To simplify the presentation, we begin by assuming that Φ is finite; the case of an infinite vocabulary is deferred to §2.3. Much of the material in §§2.1 and 2.2 is taken from [23].

2.1. The random-worlds method. We begin by defining the random-worlds, or labeled, method. Given a sentence $\xi \in \mathcal{L}(\Phi)$, let $\#world_N^{\Phi}(\xi)$ be the number of worlds, or first-order models, of ξ over Φ with domain $\{1, \ldots, N\}$. Note that the assumption that Φ is finite is necessary for $\#world_N^{\Phi}(\xi)$ to be well defined. Define

$$\Pr_N^{w,\Phi}(\varphi \mid \theta) = \frac{\#world_N^{\Phi}(\varphi \wedge \theta)}{\#world_N^{\Phi}(\theta)} .$$

In [23], we proved the following proposition.

PROPOSITION 2.1. *Let Φ, Φ' be finite vocabularies, and let φ, θ be sentences in both $\mathcal{L}(\Phi)$ and $\mathcal{L}(\Phi')$. Then $\Pr_N^{w,\Phi}(\varphi \mid \theta) = \Pr_N^{w,\Phi'}(\varphi \mid \theta)$.*

Thus, the value of $\Pr_N^{w,\Phi}(\varphi \mid \theta)$ does not depend on the choice of Φ. We therefore omit reference to Φ in $\Pr_N^{w,\Phi}(\varphi \mid \theta)$, writing $\Pr_N^{w}(\varphi \mid \theta)$ instead.

We would like to define $\Pr_\infty^{w}(\varphi \mid \theta)$ as the limit $\lim_{N\to\infty} \Pr_N^{w}(\varphi \mid \theta)$. There is a small technical problem we have to deal with in this definition: we must decide what to do if $\#world_N^{\Phi}(\theta) = 0$, so that $\Pr_N^{w}(\varphi \mid \theta)$ is not well defined. In [23], we differentiate between the case where $\Pr_N^{w}(\varphi \mid \theta)$ is well defined for all but finitely many N's, and the case where it is well defined for infinitely many N's. As we shall show (see Lemma 3.30) this distinction need not be made when θ is a unary formula. Thus, for the purposes of this paper, we use the following definition of well-definedness, which is simpler than that of [23].

DEFINITION 2.2. The asymptotic conditional probability according to the random-worlds method, denoted $\Pr_\infty^{w}(\varphi \mid \theta)$, is *well defined* if $\#world_N^{\Phi}(\theta) \neq 0$ for all but finitely many N. If $\Pr_\infty^{w}(\varphi \mid \theta)$ is well defined, then we take $\Pr_\infty^{w}(\varphi \mid \theta)$ to denote $\lim_{N\to\infty} \Pr_N^{w}(\varphi \mid \theta)$. □

Note that for any formula φ, the issue of whether $\Pr_\infty^{w}(\varphi \mid \theta)$ is well defined is completely determined by θ. Therefore, when investigating the question of how to decide whether such a probability is well defined, it is often useful to ignore φ. We therefore say that $\Pr_\infty^{w}(* \mid \theta)$ *is well defined* if $\Pr_\infty^{w}(\varphi \mid \theta)$ is well defined for every formula φ (which is true iff $\Pr_\infty^{w}(\mathit{true} \mid \theta)$ is well defined).

2.2. The random-structures method. As we explained in the introduction, the random-structures method is motivated by the intuition that worlds that are indistinguishable within the language should only be counted once. Thus, the random-structures method counts the number of (*unlabeled*) *structures*, or isomorphism classes of worlds.

Formally, we proceed as follows. Given a sentence $\xi \in \mathcal{L}(\Phi)$, let $\#\mathit{struct}_N^{\Phi}(\xi)$ be the number of isomorphism classes of worlds with domain $\{1, \ldots, N\}$ over the vocabulary Φ satisfying ξ. Note that since all the worlds that make up a structure agree on the truth value they assign to ξ, it makes sense to talk about a structure satisfying or not satisfying ξ. We can then proceed, as before, to define $\Pr_N^{s,\Phi}(\varphi \mid \theta)$ as $\frac{\#\mathit{struct}_N^{\Phi}(\varphi\wedge\theta)}{\#\mathit{struct}_N^{\Phi}(\theta)}$. We define asymptotic conditional probability, denoted $\Pr_\infty^{s,\Phi}(\varphi \mid \theta)$ in terms of $\Pr_N^{s,\Phi}(\varphi \mid \theta)$, in analogy to the earlier definition for random worlds. It is clear that $\#\mathit{world}_N^{\Phi}(\theta) = 0$ iff $\#\mathit{struct}_N^{\Phi}(\theta) = 0$, so that well-definedness is equivalent for the two methods, for any φ, θ.

PROPOSITION 2.3. *For any* $\theta \in \mathcal{L}(\Phi)$, $\Pr_\infty^{w}(* \mid \theta)$ *is well defined iff* $\Pr_\infty^{s,\Phi}(* \mid \theta)$ *is well defined.*

As the following example, taken from [23], shows, for the random-structures method the analogue to Proposition 2.1 does not hold; the value of $\Pr_N^{s,\Phi}(\varphi \mid \theta)$, and even the value of the limit, depends on the choice of Φ. This example, together with Proposition 2.1, also demonstrates that the values of conditional probabilities generally differ between the random-worlds method and the random-structures method. By way of contrast, Fagin [14] showed that the random-worlds and random-structures methods give the same answers for unconditional probabilities, if we do not have constant or function symbols in the language.

Example 2.4. Suppose $\Phi = \{P\}$, where P is a unary predicate symbol. Let θ be $\exists! x\, P(x) \vee \neg\exists x\, P(x)$ (where, as usual, "$\exists!$" means "exists a unique"), and let φ be $\exists x\, P(x)$. For any domain size N, $\#\mathit{struct}_N^{\Phi}(\theta) = 2$. In one structure, there is exactly one element satisfying P and $N - 1$ satisfying $\neg P$; in the other, all elements satisfy $\neg P$. Therefore, $\Pr_\infty^{s,\Phi}(\varphi \mid \theta) = \frac{1}{2}$.

Now, consider $\Phi' = \{P, Q\}$, for a new unary predicate Q. There are $2N$ structures where there exists an element satisfying P: the element satisfying P may or may not satisfy Q, and of the $N - 1$ elements satisfying $\neg P$, any number between 0 and $N - 1$ may also satisfy Q. On the other hand, there are $N + 1$ structures where all elements satisfy $\neg P$: any number of elements between 0 and N may satisfy Q. Therefore, $\Pr_N^{s,\Phi'}(\varphi \mid \theta) = \frac{2N}{3N+1}$, and $\Pr_\infty^{s,\Phi'}(\varphi \mid \theta) = \frac{2}{3}$.

We know that the asymptotic limit for the random-worlds method will be the same, whether we use Φ or Φ'. Using Φ, notice that the single structure where $\exists! x\, P(x)$ is true contains N worlds (corresponding to the choice of element satisfying P), whereas the other possible structure contains only one world. Therefore, $\Pr_\infty^{w}(\varphi \mid \theta) = 1$. □

Although the two methods give different answers in general, we shall see in the next section that there are important circumstances under which they agree.

2.3. Infinite vocabularies. Up to now we have assumed that the vocabulary Φ is finite. As we observed, this assumption is crucial in our definitions of $\#\mathit{world}_N^{\Phi}(\xi)$ and $\#\mathit{struct}_N^{\Phi}(\xi)$. Nevertheless, in many standard complexity arguments it is important that the vocabulary be infinite. For example, satisfiability for propositional logic formulas is decidable in linear time if we consider a single finite vocabulary; we need to consider the class of formulas definable over some infinite vocabulary of propositional symbols to get NP-completeness.

How can we modify the random-worlds and random-structures methods to deal with an infinite vocabulary Ω? The issue is surprisingly subtle. One plausible choice depends on the observation that even if Ω is infinite, the set of symbols appearing in a given sentence is always finite. We can thus do our computations relative to this set. More formally, if $\Omega_{\varphi\wedge\theta}$ denotes the set of symbols in Ω that actually appear in $\varphi \wedge \theta$, we could define $\Pr_N^{w,\Omega}(\varphi \mid \theta) = \Pr_N^{w,\Omega_{\varphi\wedge\theta}}(\varphi \mid \theta)$. Similarly, for the random-structures method, we could define $\Pr_N^{s,\Omega}(\varphi \mid \theta) = \Pr_N^{s,\Omega_{\varphi\wedge\theta}}(\varphi \mid \theta)$. The problem with this approach is that the values given by the random-structures approach depend on the vocabulary, and it is easy to find two equivalent sentences φ and φ' such that $\Omega_\varphi \neq \Omega_{\varphi'}$ and $\Pr_\infty^{s,\Omega_{\varphi\wedge\theta}}(\varphi \mid \theta) \neq \Pr_\infty^{s,\Omega_{\varphi'\wedge\theta}}(\varphi' \mid \theta)$. (A simple example of this phenomenon can be obtained by modifying Example 2.4 slightly.) Thus, under this approach, the value of asymptotic conditional probabilities can depend on the precise syntax of the sentences involved. We view this as undesirable, and so we focus on the following two interpretations of infinite vocabularies.[3]

The first of these two alternative approaches treats an infinite vocabulary as a limit of finite subvocabularies. Assume for ease of exposition that Ω is countable. Let Ω_m consist of the first m symbols in Ω (using some arbitrary fixed ordering). We can then define $\Pr_N^{w,\Omega}(\varphi \mid \theta) = \lim_{m\to\infty} \Pr_N^{w,\Omega_m}(\varphi \mid \theta)$ (where we take $\Pr_N^{w,\Omega_m}(\varphi \mid \theta)$ to be undefined if $\varphi, \theta \notin \mathcal{L}(\Omega_m)$).[4] Similarly, we can define $\Pr_N^{s,\Omega}(\varphi \mid \theta) = \lim_{m\to\infty} \Pr_N^{s,\Omega_m}(\varphi \mid \theta)$. It follows from the results we prove below that these limits are independent of the ordering of the symbols in the vocabulary.

The second interpretation is quite different. In it, although there may be an infinite vocabulary Ω in the background, we assume that each problem instance comes along with a finite vocabulary Φ as part of the input. Thus, in our infinite vocabulary Ω, we may have predicates that are relevant to medical diagnosis, physics experiments, automobile insurance, etc. When thinking about medical applications, we use that finite portion Φ of the infinite vocabulary that is appropriate. In this approach, we always deal with finite vocabularies, but ones whose size is potentially unbounded because we do not fix the relevant vocabulary in advance.

In essence, the first approach can be viewed as saying that there really is an infinite vocabulary, while the second approach considers there to be an infinite collection of finite vocabularies (with no bound on the size of the vocabularies in the collection). The distinction between these possibilities is not usually examined as closely as we have done here. This is because the difference is rarely important. For example, propositional satisfiability is NP-complete over an infinite vocabulary, no matter how we interpret "infinite." In our context, the difference turns out to be moderately significant. For random worlds, an argument based on Proposition 2.1 shows the two approaches lead to the same answers (as does the approach that we rejected where, when computing $\Pr_N^{w,\Omega}(\varphi \mid \theta)$, we restrict the vocabulary to $\Omega_{\varphi\wedge\theta}$). On the other hand, the two approaches can lead to quite different answers in the case of the random-structures approach. It is important to point out, however, that the complexity of all problems we consider turns out to be the same no matter which interpretation of "infinite" we use.

In fact, as we now show, according to the first approach the random-structures method and the random-worlds method agree whenever we have an infinite vocabulary (and thus we have an analogue to Fagin's result [14] for the case of unconditional probabilities). A structure of size N is an equivalence class of at most $N!$ worlds, since there are at most $N!$ worlds isomorphic to a given world. We say that such a structure is *rigid* if it consists of exactly $N!$ worlds. It is easy to see that a structure is rigid just if every (nontrivial) permutation of the domain elements in a world that makes up the structure produces a *different* world in that structure. We say a world is rigid if the corresponding structure is.

Example 2.5. Let Φ consist of a single unary predicate P, and consider the worlds over the domain $\{1, 2, 3\}$. All worlds where the denotation of P contains exactly two elements are isomorphic. Therefore, these worlds form a single structure $\mathcal{S}$. There are three worlds in

[3] We note, however, that all our later complexity results concerning infinite vocabularies can be easily shown to hold for the definition just discussed.

[4] Here, we chose to take the limit on the vocabulary, and only then to take the limit on the domain size. We could, however, have chosen to reverse the order of the limits, or to consider arbitrary joint limits of these two parameters. The approach taken here seems to be the most well motivated in this framework.

$\mathcal{S}$, corresponding to the possible denotations of P: $\{1, 2\}$, $\{1, 3\}$, $\{2, 3\}$. Therefore, $\mathcal{S}$ is not rigid. In fact, it is easy to see that no structure over Φ is rigid. Now, consider structures over $\Phi' = \{P, Q\}$, where Q is a new unary predicate. The set of all worlds where the denotation of P contains two elements no longer forms a structure over Φ'. For example, one structure $\mathcal{S}'$ over Φ' is the set of worlds where the denotations of $P \wedge Q$, $P \wedge \neg Q$, and $\neg P \wedge Q$ each contain one element. There are six worlds in $\mathcal{S}'$, corresponding to the possible permutations of the three domain elements. Therefore, $\mathcal{S}'$ is rigid. □

This example demonstrates that increasing the vocabulary tends to cause rigidity. We now formalize this intuition, and show its importance. Note that in the following definition (and throughout the paper) all logarithms are taken to the base 2.

DEFINITION 2.6. We say that a vocabulary Φ is *sufficiently rich* with respect to N if

(a) Φ contains at least κ_N constant symbols and $\kappa_N \geq N^2 \log N$, or

(b) Φ contains at least π_N unary predicate symbols and $\pi_N \geq 3 \log N$, or

(c) Φ contains at least one nonunary predicate symbol. □

Fagin showed that if Φ contains at least one nonunary predicate symbol, then the number of worlds over Φ of size N is asymptotically $N!$ times the number of structures [14]. That is, almost all structures are rigid in this case. We now generalize this result. Let *rigid* be an assertion that is true only in rigid structures or rigid worlds; note that *rigid* cannot be expressed in first-order logic. If $F(N)$ and $G(N)$ are two functions of N, we write $F(N) \sim G(N)$ if $\lim_{N\to\infty} F(N)/G(N) = 1$.

THEOREM 2.7. *Suppose that for every N, Φ and Ω_N are disjoint finite vocabularies such that Ω_N is sufficiently rich with respect to N. Then for any $\xi \in \mathcal{L}(\Phi)$,*

$$\lim_{N\to\infty} \Pr_N^{s,\Phi\cup\Omega_N}(\mathit{rigid} \mid \xi) = 1,$$

provided that ξ is satisfiable for all sufficiently large domains. Hence, $\#\mathit{world}_N^{\Phi\cup\Omega_N}(\xi) \sim N!\#\mathit{struct}_N^{\Phi\cup\Omega_N}(\xi)$.

Proof. We first prove the result under the additional assumptions that $\xi = \mathit{true}$ and $\Phi = \emptyset$. We consider each of the three possibilities for sufficient richness separately, and for each case we show that almost all structures are rigid. As we said above, the case where Ω_N contains at least one nonunary predicate and $\xi = \mathit{true}$ is Fagin's result, so we need only consider the remaining two cases.

Suppose $\xi = \mathit{true}$, $\Phi = \emptyset$, and Ω_N contains κ_N constant symbols. Without loss of generality, we can assume that these constants are the only symbols in Ω_N, because any expansion of a rigid structure over Ω_N to a richer vocabulary will also be rigid. Consider a structure $\mathcal{S}$. All the worlds that make up $\mathcal{S}$ must agree on the equality relations between the interpretations of the constants. That is, for any pair of constant symbols c and c', either they are equal in all worlds that make up the structure or not equal in all of them. Thus, a lower bound on the number of distinct structures over Ω_N is given by the number of ways of partitioning κ_N objects into N or fewer equivalence classes. There is no closed form expression for this number, but a simple lower bound is obtained by counting structures where the first N constants denote distinct objects. There are $N^{(\kappa_N - N)}$ such structures, because we must choose, for each of the other constants, to which of the first N constants it is equal. It is easy to see that if all or all but one of the elements in a structure (that is, in any of the worlds in that structure) are denoted by some constant, then this structure is rigid. Hence, if a structure is nonrigid, then two or more elements are not denoted by any constant. Thus, an upper bound on the number of nonrigid structures is $(N-2)^{\kappa_N}$. Therefore,

$$\Pr_N^{s,\Omega_N}(\neg\mathit{rigid} \mid \mathit{true}) \leq \frac{(N-2)^{\kappa_N}}{N^{\kappa_N - N}} = N^N \left(1 - \frac{2}{N}\right)^{\kappa_N} < N^N e^{-2\frac{\kappa_N}{N}}.$$

This will tend to 0 if $\kappa_N \geq N^2 \log N$.

Next, suppose that $\xi = \mathit{true}$, $\Phi = \emptyset$, and Ω_N contains π_N unary predicate symbols. As before, we can assume that these predicates comprise all of Ω_N. Consider a structure $\mathcal{S}$ and a world $\mathcal{W}$ in the isomorphism class making up that structure. These π_N unary predicates partition the domain of $\mathcal{W}$ into 2^{π_N} cells, according to the subset of predicates satisfied by each of the domain elements. Notice that the predicates actually partition each of the isomorphic worlds in s in the same way (in that corresponding elements of the partition have the same size). Thus, a lower bound on the number of distinct structures over Φ is the number of ways of allocating N indistinguishable elements into 2^{π_N} distinguishable cells, which is $\binom{2^{\pi_N}+N-1}{N}$. Clearly, a structure is nonrigid if and only if some element of the partition contains more than one domain element. Thus, an upper bound on the number of nonrigid structures can be obtained by counting the number of structures over $N-1$ elements, then choosing one of the these elements to be a "double" element, representing two elements. This can be done in $(N-1)\binom{2^{\pi_N}+N-2}{N-1}$ ways. Therefore,

$$\Pr_N^{s,\Omega_N}(\neg \mathit{rigid} \mid \mathit{true}) \le \frac{(N-1)\binom{2^{\pi_N}+N-2}{N-1}}{\binom{2^{\pi_N}+N-1}{N}} = \frac{N^2-N}{2^{\pi_N}+N-1}.$$

This tends to zero if $2^{\pi_N}/N^2 \to \infty$ as N grows, which is ensured by the assumption $\pi_N \ge 3\log N$.

Finally, we drop the assumptions that $\xi = \mathit{true}$ and $\Phi = \emptyset$. Given a structure over Ω_N, we can choose the denotation for the predicates in Φ in any way that satisfies ξ. It is easy to verify that if the original structure is rigid, all such choices lead to distinct structures. Therefore,

$$\#\mathit{struct}_N^{\Phi\cup\Omega_N}(\mathit{rigid} \wedge \xi) \ge \#\mathit{struct}_N^{\Omega_N}(\mathit{rigid}) \cdot \#\mathit{world}_N^{\Phi}(\xi)\ .$$

On the other hand, clearly

$$\#\mathit{struct}_N^{\Phi\cup\Omega_N}(\neg\mathit{rigid} \wedge \xi) \le \#\mathit{struct}_N^{\Omega_N}(\neg\mathit{rigid}) \cdot \#\mathit{world}_N^{\Phi}(\xi)\ .$$

The second factor is the same in both these bounds, and therefore

$$\Pr_N^{s,\Phi\cup\Omega_N}(\mathit{rigid} \mid \xi) \ge \Pr_N^{s,\Omega_N}(\mathit{rigid} \mid \mathit{true})\ .$$

From our results for $\xi = \mathit{true}$ and $\Phi = \emptyset$ we conclude that $\lim_{N\to\infty} \Pr_N^{s,\Phi\cup\Omega_N}(\mathit{rigid} \mid \xi) = 1$. □

We also need to prove an analogous result for the random-worlds method. Note that while, if we restrict to formulas in $\mathcal{L}(\Phi)$, the answers given by the random-worlds method are independent of the vocabulary, the predicate *rigid* has a special definition in terms of the random-structures method, and so rigidity may well depend on the vocabulary. Thus, in the next result, we are careful to mention the vocabulary being used.

COROLLARY 2.8. *Suppose that for every N, Φ and Ω_N are disjoint finite vocabularies such that Ω_N is sufficiently rich with respect to N. Then for any $\xi \in \mathcal{L}(\Phi)$,*

$$\lim_{N\to\infty} \Pr_N^{w,\Phi\cup\Omega_N}(\mathit{rigid} \mid \xi) = 1,$$

provided that ξ is satisfiable in all sufficiently large domains.

Proof. Any rigid structure with domain size N that satisfies ξ corresponds to $N!$ worlds. On the other hand, nonrigid structures correspond to fewer than $N!$ worlds. It follows that the proportion of worlds satisfying ξ that are rigid is at least as great as the proportion of structures satisfying ξ that are rigid. Since the latter proportion is asymptotically 1, so is the former. □

Our main use of this theorem is in the following two corollaries. The first shows that when the vocabulary is infinite (and therefore sufficiently rich) the random-worlds and random-structures methods coincide. The second corollary shows that the same thing happens when the vocabulary is sufficiently rich because of a high-arity predicate, as long as this predicate does not appear in the formula we are conditioning on.

COROLLARY 2.9. *Suppose that Ω is infinite and $\varphi, \theta \in \mathcal{L}(\Omega)$. Then*

(a) $\Pr_N^{w,\Omega}(\varphi \mid \theta) \sim \Pr_N^{s,\Omega}(\varphi \mid \theta)$,

(b) $\Pr_\infty^{w,\Omega}(\varphi \mid \theta) = \Pr_\infty^{s,\Omega}(\varphi \mid \theta)$.

Proof. Fix N, and let Ω_m be the first m symbols in some enumeration of Ω. We will be interested in the limit as $m \longrightarrow \infty$, so without loss of generality assume that $m > N^2 \log N + |\Omega_{\varphi\wedge\theta}|$. Clearly $\Omega_m - \Omega_{\varphi\wedge\theta}$ is sufficiently rich with respect to N, so by Theorem 2.7, almost all structures are rigid. Since a rigid structure over a domain of size N consists of $N!$ worlds, we get:

$$\Pr_N^{w,\Omega_{\varphi\wedge\theta}\cup\Omega_m}(\varphi \mid \theta) = \frac{\#world_N^{\Omega_{\varphi\wedge\theta}\cup\Omega_m}(\varphi \wedge \theta)}{\#world_N^{\Omega_{\varphi\wedge\theta}\cup\Omega_m}(\theta)} \sim \frac{\#struct_N^{\Omega_{\varphi\wedge\theta}\cup\Omega_m}(\varphi \wedge \theta)}{\#struct_N^{\Omega_{\varphi\wedge\theta}\cup\Omega_m}(\theta)}$$
$$= \Pr_N^{s,\Omega_{\varphi\wedge\theta}\cup\Omega_m}(\varphi \mid \theta).$$

Since this holds for any sufficiently large m, it certainly holds at the limit. This proves part (a). Part (b) follows easily. □

We can easily strengthen part (a) and prove that we actually have $\Pr_N^{w,\Omega}(\varphi \mid \theta) = \Pr_N^{s,\Omega}(\varphi \mid \theta)$, for all N. Since we do not need this result in this paper, we omit the proof here. We remark that this result also holds for much richer languages; we did not use the fact that we were dealing with first-order logic anywhere in the proof.

COROLLARY 2.10. *Suppose that $\varphi, \theta \in \mathcal{L}(\Phi)$ where Φ contains some nonunary predicate symbol that does not appear in θ. Then* $\Pr_\infty^{w}(\varphi \mid \theta) = \Pr_\infty^{s,\Phi}(\varphi \mid \theta)$.

Proof. Using the rules of probability theory, we know that

$$\Pr_\infty^{s,\Phi}(\varphi \mid \theta) = \Pr_\infty^{s,\Phi}(\varphi \mid \theta\wedge rigid)\cdot\Pr_\infty^{s,\Phi}(rigid \mid \theta) + \Pr_\infty^{s,\Phi}(\varphi \mid \theta\wedge\neg rigid)\cdot\Pr_\infty^{s,\Phi}(\neg rigid \mid \theta),$$

if all limits exist. Because of the high-arity predicate, $\Phi - \Phi_\theta$ is sufficiently rich with respect to any N. Therefore, by Theorem 2.7, we deduce that $\Pr_\infty^{s,\Phi}(rigid \mid \theta) = 1$ and $\Pr_\infty^{s,\Phi}(\neg rigid \mid \theta) = 0$. Thus

$$\Pr_\infty^{s,\Phi}(\varphi \mid \theta) = \Pr_\infty^{s,\Phi}(\varphi \mid \theta \wedge rigid) .$$

Using Corollary 2.8 instead of Theorem 2.7, we can similarly show

$$\Pr_\infty^{w,\Phi}(\varphi \mid \theta) = \Pr_\infty^{w,\Phi}(\varphi \mid \theta \wedge rigid) .$$

Because of rigidity,

$$\Pr_\infty^{s,\Phi}(\varphi \mid \theta \wedge rigid) = \Pr_\infty^{w,\Phi}(\varphi \mid \theta \wedge rigid).$$

The result now follows immediately. □

3. Asymptotic probabilities. We begin by defining some notation that will be used consistently throughout the rest of the paper. We use Φ to denote a finite vocabulary, which may include nonunary as well as unary predicate symbols and constant symbols. We take $\mathcal{P}$ to be the set of all unary predicates in Φ, $\mathcal{C}$ to be the set of all constant symbols in Φ, and define $\Psi = \mathcal{P} \cup \mathcal{C}$. Finally, if φ is a formula, we use Φ_φ to denote those symbols in Φ that appear in φ; we can similarly define $\mathcal{C}_\varphi$, $\mathcal{P}_\varphi$, and Ψ_φ.

Our goal is to show how to compute asymptotic conditional probabilities. As we explained in the introduction, the main idea is the following. To compute $\Pr^w_\infty(\varphi \mid \theta)$, we partition the models of θ into a finite collection of classes, such that φ behaves *uniformly* in any individual class, that is, there is a 0-1 law for the asymptotic probability of φ when we restrict attention to models in a single class. Computing $\Pr^w_\infty(\varphi \mid \theta)$ reduces to first identifying the classes, computing the relative weight of each class (which is required because the classes are not necessarily of equal relative size), and then deciding, for each class, whether the asymptotic probability of φ is zero or one. In this section we deal with the logical aspects of this process; namely, showing how to construct an appropriate partition into classes. In the next section, we use results from this section to construct algorithms that compute asymptotic probabilities, and examine the complexity of these algorithms.

For most of this section, we will concentrate on the asymptotic probability according to random worlds. In §3.5 we discuss the modifications needed to deal with random structures, which are relatively minor.

3.1. Unary vocabularies and atomic descriptions. The success of the approach outlined above depends on the lack of expressivity of unary languages. In this section we show that sentences in $\mathcal{L}(\Psi)$ can only assert a fairly limited class of constraints. For instance, one corollary of our general result will be the well-known theorem that, if $\theta \in \mathcal{L}(\Psi)$ is satisfiable at all, it is satisfiable in a "small" model, one of size at most exponential in the size of the θ. (See [1] for a proof of this result and further historical references.)

We start with some definitions.

DEFINITION 3.1. Given a vocabulary Φ and a finite set of variables $\mathcal{X}$, a *complete description D* over Φ and $\mathcal{X}$ is an unquantified conjunction of formulas such that

- for every predicate $R \in \Phi \cup \{=\}$ of arity m, and for every $z_1, \ldots, z_m \in \mathcal{C} \cup \mathcal{X}$, D contains exactly one of $R(z_1, \ldots, z_m)$ or $\neg R(z_1, \ldots, z_m)$ as a conjunct;
- D is consistent.[5] □

We can think of a complete description as being a formula that describes as fully as possible the behavior of the predicate symbols in Φ over the constant symbols in Φ and the variables in $\mathcal{X}$.

We can also consider complete descriptions over subsets of Φ. The case when we look just at the unary predicates and a single variable x will be extremely important.

DEFINITION 3.2. Let $\mathcal{P}$ be $\{P_1, \ldots, P_k\}$. An *atom* over $\mathcal{P}$ is a complete description over $\mathcal{P}$ and some variable $\{x\}$. More precisely, it is a conjunction of the form $P'_1(x) \wedge \ldots \wedge P'_k(x)$, where each P'_i is either P_i or $\neg P_i$. Since the variable x is irrelevant to our concerns, we typically suppress it and describe an atom as a conjunction of the form $P'_1 \wedge \ldots \wedge P'_k$. □

Note that there are $2^k = 2^{|\mathcal{P}|}$ atoms over $\mathcal{P}$, and that they are mutually exclusive and exhaustive. We use $A_1, \ldots, A_{2^{|\mathcal{P}|}}$ to denote the atoms over $\mathcal{P}$, listed in some fixed order. For example, there are four atoms over $\mathcal{P} = \{P_1, P_2\}$: $A_1 = P_1 \wedge P_2$, $A_2 = P_1 \wedge \neg P_2$, $A_3 = \neg P_1 \wedge P_2$, $A_4 = \neg P_1 \wedge \neg P_2$.

We now want to define the notion of *atomic description* which is, roughly speaking, a maximally expressive formula in the unary vocabulary Ψ. Fix a natural number M. A size M atomic description consists of two parts. The first part, the *size description with bound M*, specifies exactly how many elements in the domain should satisfy each atom A_i, except that if there are M or more elements satisfying the atom it only expresses that fact, rather than giving the exact count. More formally, given a formula $\xi(x)$ with a free variable x, we take $\exists^m x\, \xi(x)$ to be the sentence that says there are precisely m domain elements satisfying ξ:

[5] Inconsistency is possible because of the use of equality. For example, if D includes $z_1 = z_2$ as well as both $R(z_1, z_3)$ and $\neg R(z_2, z_3)$, it is inconsistent.

$$\exists^m x\, \xi(x) =_{\text{def}} \exists x_1 \ldots x_m \left(\bigwedge_i \left(\xi(x_i) \wedge \bigwedge_{j \neq i} (x_j \neq x_i) \right) \wedge \forall y (\xi(y) \Rightarrow \vee_i (y = x_i)) \right).$$

Similarly, we define $\exists^{\geq m} x\, \xi(x)$ to be the formula that says that there are at least m domain elements satisfying ξ:

$$\exists^{\geq m} x\, \xi(x) =_{\text{def}} \exists x_1 \ldots x_m \left(\bigwedge_i \left(\xi(x_i) \wedge \bigwedge_{j \neq i} (x_j \neq x_i) \right) \right).$$

DEFINITION 3.3. A *size description with bound M (over $\mathcal{P}$)* is a conjunction of $2^{|\mathcal{P}|}$ formulas: for each atom A_i over $\mathcal{P}$, it includes either $\exists^{\geq M} x\, A_i(x)$ or a formula of the form $\exists^m x\, A_i(x)$ for some $m < M$. □

The second part of an atomic description is a complete description that specifies the properties of constants and free variables.

DEFINITION 3.4. A *size M atomic description* (over Ψ and $\mathcal{X}$) is a conjunction of:

- a size description with bound M over $\mathcal{P}$, and
- a complete description over Ψ and $\mathcal{X}$. □

Note that an atomic description is a finite formula, and there are only finitely many size M atomic descriptions over Ψ and $\mathcal{X}$ (for fixed M). For the purposes of counting atomic descriptions (as we do in §3.4), we assume some arbitrary but fixed ordering of the conjuncts in an atomic description. Under this assumption, we cannot have two distinct atomic descriptions that differ only in the ordering of conjuncts. Given this, it is easy to see that atomic descriptions are mutually exclusive. Moreover, atomic descriptions are exhaustive—the disjunction of all consistent atomic descriptions of size M is valid.

Example 3.5. Consider the following size description σ with bound 4 over $\mathcal{P} = \{P_1, P_2\}$:

$$\exists^1 x\, A_1(x) \wedge \exists^3 x\, A_2(x) \wedge \exists^{\geq 4} x\, A_3(x) \wedge \exists^{\geq 4} x\, A_4(x).$$

Let $\Psi = \{P_1, P_2, c_1, c_2, c_3\}$. It is possible to augment σ into an atomic description in many ways. For example, one consistent atomic description ψ_* of size 4 over Ψ and $\emptyset$ (no free variables) is:[6]

$$\sigma \wedge A_2(c_1) \wedge A_3(c_2) \wedge A_3(c_3) \wedge c_1 \neq c_2 \wedge c_1 \neq c_3 \wedge c_2 = c_3.$$

On the other hand, the atomic description

$$\sigma \wedge A_1(c_1) \wedge A_1(c_2) \wedge A_3(c_3) \wedge c_1 \neq c_2 \wedge c_1 \neq c_3 \wedge c_2 \neq c_3$$

is an inconsistent atomic description, since σ dictates that there is precisely one element in the atom A_1, whereas the second part of the atomic description implies that there are two distinct domain elements in that atom. □

As we explained, an atomic description is, intuitively, a maximally descriptive sentence over a unary vocabulary. The following theorem formalizes this idea by showing that each unary formula is equivalent to a disjunction of atomic descriptions. For a given M and set $\mathcal{X}$ of variables, let $\mathcal{A}^{\Psi}_{M,\mathcal{X}}$ be the set of consistent atomic descriptions of size M over Ψ and $\mathcal{X}$.

DEFINITION 3.6. Let $d(\xi)$ denote the *depth of quantifier nesting* in ξ. We define $d(\xi)$ by induction on the structure of ξ as follows:

[6] In our examples, we use the commutativity of equality in order to avoid writing down certain superfluous disjuncts. In this example, for instance, we do not write down both $c_1 \neq c_2$ and $c_2 \neq c_1$.

- $d(\xi) = 0$ for any atomic formula ξ,
- $d(\neg\xi) = d(\xi)$,
- $d(\xi_1 \wedge \xi_2) = d(\xi_1 \vee \xi_2) = \max(d(\xi_1), d(\xi_2))$,
- $d(\forall y\, \xi) = d(\exists y\, \xi) = d(\xi) + 1$. □

THEOREM 3.7. *If ξ is a formula in $\mathcal{L}(\Psi)$ whose free variables are contained in $\mathcal{X}$, and $M \geq d(\xi) + |\mathcal{C}| + |\mathcal{X}|$, then there exists a set of atomic descriptions $\mathcal{A}^{\Psi}_{\xi} \subseteq \mathcal{A}^{\Psi}_{M,\mathcal{X}}$ such that ξ is equivalent to $\bigvee_{\psi \in \mathcal{A}^{\Psi}_{\xi}} \psi$.*

Proof. We proceed by a straightforward induction on the structure of ξ. We assume without loss of generality that ξ is constructed from atomic formulas using only the operators $\wedge$, $\neg$, and $\exists$.

First suppose that ξ is an atomic formula. That is, ξ is either of the form $P(z)$ or of the form $z = z'$, for $z, z' \in \mathcal{C} \cup \mathcal{X}$. In this case, either the formula ξ or its negation appears as a conjunct in each atomic description $\psi \in \mathcal{A}^{\Psi}_{M,\xi}$. Let $\mathcal{A}^{\Psi}_{\xi}$ be those atomic descriptions in which ξ appears as a conjunct. Clearly, ξ is inconsistent with the remaining atomic descriptions. Since the disjunction of the atomic descriptions in $\mathcal{A}^{\Psi}_{M,\mathcal{X}}$ is valid, we obtain that ξ is equivalent to $\bigvee_{\psi \in \mathcal{A}^{\Psi}_{\xi}} \psi$.

If ξ is of the form $\xi_1 \wedge \xi_2$, then by the induction hypothesis, ξ_i is equivalent to the disjunction of a set $\mathcal{A}^{\Psi}_{\xi_i} \subseteq \mathcal{A}^{\Psi}_{M,\mathcal{X}}$, for $i = 1, 2$. Clearly ξ is equivalent to the disjunction of the atomic descriptions in $\mathcal{A}^{\Psi}_{\xi_1} \cap \mathcal{A}^{\Psi}_{\xi_2}$. (Recall that the empty disjunction is equivalent to the formula *false*.)

If ξ is of the form $\neg\xi'$ then, by the induction hypothesis, ξ' is equivalent to the disjunction of the atomic descriptions in $\mathcal{A}^{\Psi}_{\xi'}$. It is easy to see that ξ is the disjunction of the atomic descriptions in $\mathcal{A}^{\Psi}_{\neg\xi'} = \mathcal{A}^{\Psi}_{M,\mathcal{X}} - \mathcal{A}^{\Psi}_{\xi'}$.

Finally, we consider the case that ξ is of the form $\exists y\, \xi'$. Recall that $M \geq d(\xi) + |\mathcal{C}| + |\mathcal{X}|$. Since $d(\xi') = d(\xi) - 1$, it is also the case that $M \geq d(\xi') + |\mathcal{C}| + |\mathcal{X} \cup \{y\}|$. By the induction hypothesis, ξ' is therefore equivalent to the disjunction of the atomic descriptions in $\mathcal{A}^{\Psi}_{\xi'}$. Clearly ξ is equivalent to $\exists y \vee_{\psi \in \mathcal{A}^{\Psi}_{\xi'}} \psi$, and standard first-order reasoning shows that $\exists y \vee_{\psi \in \mathcal{A}^{\Psi}_{\xi'}} \psi$ is equivalent to $\vee_{\psi \in \mathcal{A}^{\Psi}_{\xi'}} \exists y\, \psi$. Since $\mathcal{A}^{\Psi}_{\xi'} \subseteq \mathcal{A}^{\Psi}_{M,\mathcal{X} \cup \{y\}}$, it suffices to show that for each atomic description $\psi \in \mathcal{A}^{\Psi}_{M,\mathcal{X} \cup \{y\}}$, $\exists y\, \psi$ is equivalent to an atomic description in $\mathcal{A}^{\Psi}_{M,\mathcal{X}}$.

Consider some $\psi \in \mathcal{A}^{\Psi}_{M,\mathcal{X} \cup \{y\}}$; we can clearly pull out of the scope of $\exists y$ all the conjuncts in ψ that do not involve y. It follows that $\exists y\, \psi$ is equivalent to $\psi' \wedge \exists y\, \psi''$, where ψ'' is a conjunction of $A(y)$, where A is an atom over $\mathcal{P}$, and formulas of the form $y = z$ and $y \neq z$. It is easy to see that ψ' is a consistent atomic description over Ψ and $\mathcal{X}$ of size M. To complete the proof, we now show that $\psi' \wedge \exists y\, \psi''$ is equivalent to ψ'. There are two cases to consider. First suppose that ψ'' contains a conjunct of the form $y = z$. Let $\psi''[y/z]$ be the result of replacing all free occurrences of y in ψ'' by z. Standard first-order reasoning (using the fact that $\psi''[y/z]$ has no free occurrences of y) shows that $\psi''[y/z]$ is equivalent to $\exists y\, \psi''[y/z]$, which is equivalent to $\exists y\, \psi''$. Since ψ is a complete atomic description which is consistent with ψ'', it follows that each conjunct of $\psi''[y/z]$ (except $z = z$) must be a conjunct of ψ', so ψ' implies $\psi''[y/z]$. It immediately follows that ψ' is equivalent to $\psi' \wedge \exists y\, \psi''$ in this case. Now suppose that there is no conjunct of the form $y = z$ in ψ''. In this case, $\exists y\, \psi''$ is certainly true if there exists a domain element satisfying atom A different from the denotations of all the symbols in $\mathcal{X} \cup \mathcal{C}$. Notice that ψ implies that there exists such an element, namely, the denotation of y. However, ψ' must already imply the existence of such an element since ψ' must force there to be enough elements satisfying A to guarantee the existence of such an element. (We remark that it is crucial for this last part of the argument that $M \geq |\mathcal{X}| + 1 + |\mathcal{C}|$.) Thus, we again have that ψ' is equivalent to $\psi' \wedge \exists y\, \psi''$. It follows that $\exists y\, \psi$ is equivalent to a consistent atomic description in $\mathcal{A}^{\Psi}_{M,\mathcal{X}}$, namely ψ', as required. □

For the remainder of this paper we will be interested in sentences. Thus, we restrict attention to atomic descriptions over Ψ and the empty set of variables. Moreover, we assume that all formulas mentioned are in fact sentences, and have no free variables.

DEFINITION 3.8. For $\Psi = \mathcal{P} \cup \mathcal{C}$, and a sentence $\xi \in \mathcal{L}(\Psi)$, we define $\mathcal{A}_\xi^\Psi$ to be the set of consistent atomic descriptions of size $d(\xi) + |\mathcal{C}|$ over Ψ such that ξ is equivalent to the disjunction of the atomic descriptions in $\mathcal{A}_\xi^\Psi$. □

It will be useful for our later results to prove a simpler analogue of Theorem 3.7 for the case where the sentence ξ does not use equality or constant symbols. A *simplified atomic description over $\mathcal{P}$* is simply a size description with bound 1. Thus, it consists of a conjunction of $2^{|\mathcal{P}|}$ formulas of the form $\exists^{\geq 1} x\, A_i(x)$ or $\exists^0 x\, A_i(x)$, one for each atom over $\mathcal{P}$. Using the same techniques as those of Theorem 3.7, we can prove the following theorem.

THEOREM 3.9. *If $\xi \in \mathcal{L}^-(\mathcal{P})$, then ξ is equivalent to a disjunction of simplified atomic descriptions over $\mathcal{P}$.*

Proof. The proof is left to the reader. □

3.2. Named elements and model descriptions. Recall that we are attempting to divide the worlds satisfying θ into classes such that:

- φ is uniform in each class, and
- the relative weight of the classes is easily computed.

In the previous section, we defined the concept of atomic description, and showed that a sentence $\theta \in \mathcal{L}(\Psi)$ is equivalent to some disjunction of atomic descriptions. This suggests that atomic descriptions might be used to classify models of θ. Līogon'kīĭ [31] has shown that this is indeed a successful approach, as long as we consider languages without constants and condition only on sentences that do not use equality. In Theorem 3.9 we showed that, for such languages, each sentence is equivalent to the disjunction of simplified atomic descriptions. The following theorem, due to Līogon'kīĭ, says that classifying models according to which simplified atomic description they satisfy leads to the desired uniformity property. This result will be a corollary of a more general theorem that we prove later.

PROPOSITION 3.10. [31] *Suppose that $\mathcal{C} = \emptyset$. If $\varphi \in \mathcal{L}(\Phi)$ and ψ is a consistent simplified atomic description over $\mathcal{P}$, then $\Pr_\infty^w(\varphi \mid \psi)$ is either 0 or 1.*

If $\mathcal{C} \neq \emptyset$, then we do not get an analogue to Proposition 3.10 if we simply partition the worlds according to the atomic description they satisfy. For example, consider the atomic description ψ_* from Example 3.5, and the sentence $\varphi = R(c_1, c_1)$ for some binary predicate R. Clearly, by symmetry, $\Pr_\infty^w(\varphi \mid \psi_*) = 1/2$, and therefore φ is not uniform over the worlds satisfying ψ_*. We do not even need to use constant symbols, such as c_1, to construct such counterexamples. Recall that the size description in ψ_* included the conjunct $\exists^1 x\, A_1(x)$. So if $\varphi' = \exists x\, (A_1(x) \wedge R(x, x))$ then we also get $\Pr_\infty^w(\varphi' \mid \psi_*) = 1/2$.

The general problem is that, given ψ_*, φ can refer "by name" to certain domain elements and thus its truth can depend on their properties. In particular, φ can refer to domain elements that are denotations of constants in $\mathcal{C}$ as well as to domain elements that are the denotations of the "fixed-size" atoms—those atoms whose size is fixed by the atomic description. In the example above, we can view "the x such that $A_1(x)$" as a name for the unique domain element satisfying atom A_1. In any model of ψ_*, we call the denotations of the constants and elements of the fixed-size atoms the *named elements* of that model. The discussion above indicates that there is no uniformity theorem if we condition only on atomic descriptions, because an atomic expression does not fix the denotations of the nonunary predicates with respect to the named elements. This analysis suggests that we should augment an atomic description with complete information about the named elements. This leads to a finer classification of models which does have the uniformity property. To define this classification formally, we need the following definitions.

DEFINITION 3.11. The *characteristic* of an atomic description ψ of size M is a tuple C_ψ of the form $\langle (f_1, g_1), \ldots, (f_{2^{|\mathcal{P}|}}, g_{2^{|\mathcal{P}|}}) \rangle$, where

- $f_i = m$ if exactly $m < M$ domain elements satisfy A_i according to ψ,
- $f_i = *$ if at least M domain elements satisfy A_i according to ψ,
- g_i is the number of distinct domain elements which are interpretations of elements in $\mathcal{C}$ that satisfy A_i according to ψ. □

Note that we can compute the characteristic of ψ immediately from the syntactic form of ψ.

DEFINITION 3.12. Suppose $C_\psi = \langle (f_1, g_1), \ldots, (f_{2^{|\mathcal{P}|}}, g_{2^{|\mathcal{P}|}}) \rangle$ is the characteristic of ψ. We say that an atom A_i is *active* in ψ if $f_i = *$; otherwise A_i is *passive*. Let $A(\psi)$ be the set $\{i \ : \ A_i \text{ is active in } \psi\}$. □

We can now define named elements.

DEFINITION 3.13. Given an atomic description ψ and a model $\mathcal{W}$ of ψ, the *named elements* in $\mathcal{W}$ are the elements satisfying the passive atoms and the elements that are denotations of constants.

The number of named elements in any model of ψ is

$$\nu(\psi) = \sum_{i \in A(\psi)} g_i + \sum_{i \notin A(\psi)} f_i,$$

where $C_\psi = \langle (f_1, g_1), \ldots, (f_{2^{|\mathcal{P}|}}, g_{2^{|\mathcal{P}|}}) \rangle$, as before. □

As we have discussed, we wish to augment ψ with information about the named elements. We accomplish this using the following notion of *model fragment* which is, roughly speaking, the projection of a model onto the named elements.

DEFINITION 3.14. Let $\psi = \sigma \wedge D$ for a size description σ and a complete description D over Ψ. A *model fragment* $\mathcal{V}$ for ψ is a model over the vocabulary Φ with domain $\{1, \ldots, \nu(\psi)\}$ such that:

- $\mathcal{V}$ satisfies D, and
- $\mathcal{V}$ satisfies the conjuncts in σ defining the sizes of the passive atoms. □

We can now define what it means for a model $\mathcal{W}$ to satisfy a model fragment $\mathcal{V}$.

DEFINITION 3.15. Let $\mathcal{W}$ be a model of ψ, and let $i_1, \ldots, i_{\nu(\psi)} \in \{1, \ldots, N\}$ be the named elements in $\mathcal{W}$, where $i_1 < i_2 < \cdots < i_{\nu(\psi)}$. The model $\mathcal{W}$ is said to *satisfy* the model fragment $\mathcal{V}$ if the function $F(j) = i_j$ from the domain of $\mathcal{V}$ to the domain of $\mathcal{W}$ is an isomorphism between $\mathcal{V}$ and the submodel of $\mathcal{W}$ formed by restricting to the named elements. □

Example 3.16. Consider the atomic description ψ_* from Example 3.5. Its characteristic C_{ψ_*} is $\langle (1, 0), (3, 1), (*, 1), (*, 0) \rangle$. The active atoms are thus A_3 and A_4. Note that $g_3 = 1$ because c_2 and c_3 are constrained to denote the same element. Thus, the number of named elements $\nu(\psi_*)$ in a model of ψ_* is $1 + 3 + 1 = 5$. Therefore each model fragment for ψ_* will have domain $\{1, 2, 3, 4, 5\}$. The elements in the domain will be the named elements; these correspond to the single element in A_1, the three elements in A_2, and the unique element denoting both c_2 and c_3 in A_3.

Let Φ be $\{P_1, P_2, c_1, c_2, c_3, R\}$ where R is a binary predicate symbol. One possible model fragment $\mathcal{V}_*$ for ψ_* over Φ gives the symbols in Φ the following interpretation:

$$\begin{array}{lll} c_1^{\mathcal{V}_*} = 4, & c_2^{\mathcal{V}_*} = 3, & c_3^{\mathcal{V}_*} = 3, \\ P_1^{\mathcal{V}_*} = \{1, 2, 4, 5\}, & P_2^{\mathcal{V}_*} = \{1, 3\}, & R^{\mathcal{V}_*} = \{(1, 3), (3, 4)\}. \end{array}$$

It is easy to verify that $\mathcal{V}_*$ satisfies the properties of the constants as prescribed by the description D in ψ_* as well as the two conjuncts $\exists^1 x\, A_1(x)$ and $\exists^3 x\, A_2(x)$ in the size description in ψ_*.

Now, let $\mathcal{W}$ be a world satisfying ψ_*, and assume that the named elements in $\mathcal{W}$ are $3, 8, 9, 14, 17$. Then $\mathcal{W}$ satisfies $\mathcal{V}_*$ if this 5-tuple of elements has precisely the same properties in $\mathcal{W}$ as the 5-tuple $1, 2, 3, 4, 5$ does in $\mathcal{V}_*$. □

Although a model fragment is a semantic structure, the definition of satisfaction just given also allows us to regard it as a logical assertion that is true or false in any model over Φ whose domain is a subset of the natural numbers. In the following, we use this view of a model description as an assertion frequently. In particular, we freely use assertions which are the conjunction of an ordinary first-order ψ and a model fragment $\mathcal{V}$, even though the result is not a first-order formula. Under this viewpoint it makes perfect sense to use an expression such as $\Pr_\infty^w(\varphi \mid \psi \wedge \mathcal{V})$.

DEFINITION 3.17. A *model description* augmenting ψ over the vocabulary Φ is a conjunction of ψ and a model fragment $\mathcal{V}$ for ψ over Φ. Let $\mathcal{M}^\Phi(\psi)$ be the set of model descriptions augmenting ψ. (If Φ is clear from context, we omit the subscript and write $\mathcal{M}(\psi)$ rather than $\mathcal{M}^\Phi(\psi)$.) □

It should be clear that model descriptions are mutually exclusive and exhaustive. Moreover, as for atomic descriptions, each unary sentence θ is equivalent to some disjunction of model descriptions. From this, and elementary probability theory, we conclude the following fact, which forms the basis of our techniques for computing asymptotic conditional probabilities.

PROPOSITION 3.18. *For any $\varphi \in \mathcal{L}(\Phi)$ and $\theta \in \mathcal{L}(\Psi)$*

$$\Pr_\infty^w(\varphi \mid \theta) = \sum_{\psi \in \mathcal{A}_\theta^\Psi} \sum_{(\psi \wedge \mathcal{V}) \in \mathcal{M}(\psi)} \Pr_\infty^w(\varphi \mid \psi \wedge \mathcal{V}) \cdot \Pr_\infty^w(\psi \wedge \mathcal{V} \mid \theta),$$

if all limits exist.

As we show in the next section, model descriptions have the uniformity property so the first term in the product will always be either 0 or 1.

It might seem that the use of model fragments is a needless complication and that any model fragment, in its role as a logical assertion, will be equivalent to some first-order sentence. Consider the following definition.

DEFINITION 3.19. Let $n = \nu(\psi)$. The *complete description capturing* $\mathcal{V}$, denoted $D_\mathcal{V}$, is a formula that satisfies the following:[7]

- $D_\mathcal{V}$ is a complete description over Φ and the variables $\{x_1, \ldots, x_n\}$ (see Definition 3.1),
- for each $i \neq j$, $D_\mathcal{V}$ contains a conjunct $x_i \neq x_j$, and
- $\mathcal{V}$ satisfies $D_\mathcal{V}$ when i is assigned to x_i for each $i = 1, \ldots, n$. □

Example 3.20. The complete description $D_{\mathcal{V}_*}$ capturing the model fragment $\mathcal{V}_*$ from the previous example has conjuncts such as $P_1(x_1)$, $\neg P_1(x_3)$, $R(x_1, x_3)$, $\neg R(x_1, x_2)$, and $x_4 = c_1$. □

The distinction between a model fragment and the complete description capturing it is subtle. Clearly if a model satisfies $\mathcal{V}$, then it also satisfies $\exists x_1, \ldots, x_n\, D_\mathcal{V}$. The converse is not necessarily true. A model fragment places additional constraints on which domain elements are denotations of the constants and passive atoms. For example, a model fragment might entail that, in any model over the domain $\{1, \ldots, N\}$, the denotation of constant c_1 is less than that of c_2. Clearly, no first-order sentence can assert this. The main implication of this difference is combinatorial; it turns out that counting model fragments (rather than the complete descriptions that capture them) simplifies many computations considerably. Although we typically use model fragments, there are occasions where it is important to remain within

[7] Note that there will, in general, be more than one complete description capturing $\mathcal{V}$. We choose one of them arbitrarily for $D_\mathcal{V}$.

first-order logic and use the corresponding complete descriptions instead. For instance, this is the case in the next subsection. Whenever we do this we will appeal to the following result, which is easy to prove.

PROPOSITION 3.21. *For any $\varphi \in \mathcal{L}(\Phi)$ and model description $\psi \wedge \mathcal{V}$ over Φ, we have*

$$\Pr_\infty^w(\varphi \mid \psi \wedge \mathcal{V}) = \Pr_\infty^w(\varphi \mid \psi \wedge \exists x_1, \ldots, x_{\nu(\psi)}\, D_{\mathcal{V}}).$$

Proof. The proof is left to the reader. □

3.3. A conditional 0-1 law. In the previous section, we showed how to partition θ into model descriptions. We now show that φ is uniform over each model description. That is, for any sentence $\varphi \in \mathcal{L}(\Phi)$ and any model description $\psi \wedge \mathcal{V}$, the probability $\Pr_\infty^w(\varphi \mid \psi \wedge \mathcal{V})$ is either 0 or 1. The technique we use to prove this is a generalization of Fagin's proof of the 0-1 law for first-order logic without constant or function symbols [13]. This result states that if φ is a first-order sentence in a vocabulary without constant or function symbols, then $\Pr_\infty^w(\varphi)$ is either 0 or 1.[8] It is well known that we can get asymptotic probabilities that are neither 0 nor 1 if we use constant symbols, or if we look at general conditional probabilities. However, in the special case where we condition on a model description there is still a 0-1 law. Throughout this section let $\psi \wedge \mathcal{V}$ be a fixed model description with at least one active atom, and let $n = \nu(\psi)$ be the number of named elements according to ψ.

As we said earlier, the proof of our 0-1 law is based on Fagin's proof. Like Fagin, our strategy involves constructing a theory T which, roughly speaking, states that any finite fragment of a model can be extended to a larger fragment in all possible ways. We then prove two propositions.

1. T is complete; that is, for each $\varphi \in \mathcal{L}(\Phi)$, either $T \models \varphi$ or $T \models \neg\varphi$. This result, in the case of the original 0-1 law, is due to Gaifman [16].
2. For any $\varphi \in \mathcal{L}(\Phi)$, if $T \models \varphi$ then $\Pr_\infty^w(\varphi \mid \psi \wedge \mathcal{V}) = 1$.

Using the first proposition, for any sentence φ, either $T \models \varphi$ or $T \models \neg\varphi$. Therefore, using the second proposition, either $\Pr_\infty^w(\varphi \mid \psi \wedge \mathcal{V}) = 1$ or $\Pr_\infty^w(\neg\varphi \mid \psi \wedge \mathcal{V}) = 1$. The latter case immediately implies that $\Pr_\infty^w(\varphi \mid \psi \wedge \mathcal{V}) = 0$. Thus, these two propositions suffice to prove the conditional 0-1 law.

We begin by defining several concepts which will be useful in defining the theory T.

DEFINITION 3.22. Let $\mathcal{X}' \supseteq \mathcal{X}$, let D be a complete description over Φ and $\mathcal{X}$, and let D' be a complete description over Φ and $\mathcal{X}'$. We say that D' *extends* D if every conjunct of D is a conjunct of D'. □

The core of the definition of T is the concept of an *extension axiom*, which asserts that any finite substructure can be extended to a larger structure containing one more element.

DEFINITION 3.23. Let $\mathcal{X} = \{x_1, \ldots, x_j\}$ for some k, let D be a complete description over Φ and $\mathcal{X}$, and let D' be any complete description over Φ and $\mathcal{X} \cup \{x_{j+1}\}$ that extends D. The sentence

$$\forall x_1, x_2, \ldots, x_j\ (D \Rightarrow \exists x_{j+1} D')$$

is an *extension axiom.* □

In the original 0-1 law, Fagin considered the theory consisting of all the extension axioms. In our case, we must consider only those extension axioms whose components are consistent with ψ, and which extend $D_{\mathcal{V}}$.

[8] As we noted in the introduction, the 0-1 law was first proved by Glebskĭi et al. [18]. However, it is Fagin's proof technique that we are using here.

DEFINITION 3.24. Given $\psi \wedge \mathcal{V}$, we define T to consist of $\psi \wedge \exists x_1, \ldots, x_n\, D_{\mathcal{V}}$ together with all extension axioms

$$\forall x_1, x_2, \ldots, x_j\ (D \Rightarrow \exists x_{j+1} D')$$

in which D (and hence D') extends $D_{\mathcal{V}}$ and in which D' (and hence D) is consistent with ψ. □

We have used $D_{\mathcal{V}}$ rather than $\mathcal{V}$ in this definition so that T is a first-order theory. Note that the consistency condition above is not redundant, even given that the components of an extension axiom extend $D_{\mathcal{V}}$. However, inconsistency can arise only if D' asserts the existence of a new element in some passive atom (because this would contradict the size description in ψ).

We now prove the two propositions that imply the 0-1 law.

PROPOSITION 3.25. *The theory T is complete.*

Proof. The proof is based on a result of Loś and Vaught [40] which says that any first-order theory with no finite models, such that all of its countable models are isomorphic, is complete. The theory T obviously has no finite models. The fact that all of its countable models are isomorphic follows by a standard "back and forth" argument. That is, let $\mathcal{U}$ and $\mathcal{U}'$ be countable models of T. Without loss of generality, assume that both models have the same domain $\mathcal{D} = \{1, 2, 3, \ldots\}$. We must find a mapping $F : \mathcal{D} \rightarrow \mathcal{D}$ which is an isomorphism between $\mathcal{U}$ and $\mathcal{U}'$ with respect to Φ.

We first map the named elements in both models to each other, in the appropriate way. Recall that T contains the assertion $\exists x_1, \ldots, x_n\, D_{\mathcal{V}}$. Since $\mathcal{U} \models T$, there must exist domain elements $d_1, \ldots, d_n \in \mathcal{D}$ that satisfy $D_{\mathcal{V}}$ under the model $\mathcal{U}$. Similarly, there must exist corresponding elements $d'_1, \ldots, d'_n \in \mathcal{D}$ that satisfy $D_{\mathcal{V}}$ under the model $\mathcal{U}'$. We define the mapping F so that $F(d_i) = d'_i$ for $i = 1, \ldots, n$. Since $D_{\mathcal{V}}$ is a complete description over these elements, and the two substructures both satisfy $D_{\mathcal{V}}$, they are necessarily isomorphic.

In the general case, assume we have already defined F over some j elements $\{d_1, d_2, \ldots, d_j\} \in \mathcal{D}$ so that the substructure of $\mathcal{U}$ over $\{d_1, \ldots, d_j\}$ is isomorphic to the substructure of $\mathcal{U}'$ over $\{d'_1, \ldots, d'_j\}$, where $d'_i = F(d_i)$ for $i = 1, \ldots, j$. Because both substructures are isomorphic there must be a description D that is satisfied by both. Since we began by creating a mapping between the named elements, we can assume that D extends $D_{\mathcal{V}}$. We would like to extend the mapping F so that it eventually exhausts both domains. We accomplish this by using the even rounds of the construction (the rounds where j is even) to ensure that $\mathcal{U}$ is covered, and the odd rounds to ensure that $\mathcal{U}'$ is covered. More precisely, if j is even, let d be the first element in $\mathcal{D}$ which is not in $\{d_1, \ldots, d_j\}$. There is a model description D' extending D that is satisfied by $d_1, \ldots, d_j, d$ in $\mathcal{U}$. Consider the extension axiom in T asserting that any j elements satisfying D can be extended to $j+1$ elements satisfying D'. Since $\mathcal{U}'$ satisfies this axiom, there exists an element d' in $\mathcal{U}'$ such that $d'_1, \ldots, d'_j, d'$ satisfy D'. We define $F(d) = d'$. It is clear that the substructure of $\mathcal{U}$ over $\{d_1, \ldots, d_j, d\}$ is isomorphic to the substructure of $\mathcal{U}'$ over $\{d'_1, \ldots, d'_j, d'\}$. If j is odd, we follow the same procedure, except that we find a counterpart to the first domain element (in $\mathcal{U}'$) which does not yet have a pre-image in $\mathcal{U}$. It is easy to see that the final mapping F is an isomorphism between $\mathcal{U}$ and $\mathcal{U}'$. □

PROPOSITION 3.26. *For any $\varphi \in \mathcal{L}(\Phi)$, if $T \models \varphi$ then $\Pr^w_\infty(\varphi \mid \psi \wedge \mathcal{V}) = 1$.*

Proof. We begin by proving the claim for a sentence $\xi \in T$. By the construction of T, ξ is either $\psi \wedge \exists x_1, \ldots, x_n\, D_{\mathcal{V}}$ or an extension axiom. In the first case, Proposition 3.21 trivially implies that $\Pr^w_\infty(\xi \mid \psi \wedge \mathcal{V}) = 1$. The proof for the case that ξ is an extension axiom is based on a straightforward combinatorial argument, which we briefly sketch. Recall that one of the conjuncts of ψ is a size description σ. The sentence σ includes two types of conjuncts: those of the form $\exists^m x\, A(x)$ and those of the form $\exists^{\geq M} x\, A(x)$. Let σ' be σ with the conjuncts of the second type removed. Let ψ' be the same as ψ except that σ' replaces σ. It is easy to show that $\Pr^w_\infty(\exists^{\geq M} x\, A(x) \mid \psi' \wedge \mathcal{V}) = 1$ for any active atom A, and so $\Pr^w_\infty(\psi \mid \psi' \wedge \mathcal{V}) = 1$. Since $\psi \Rightarrow \psi'$, by straightforward probabilistic arguments, it suffices to show that $\Pr^w_\infty(\xi \mid \psi' \wedge \mathcal{V}) = 1$.

Suppose ξ is an extension axiom involving D and D', where D is a complete description over $\mathcal{X} = \{x_1, \ldots, x_j\}$ and D' is a description over $\mathcal{X} \cup \{x_{j+1}\}$ that extends D. Fix a domain size N, and some particular j domain elements $d_1, \ldots, d_j$ that satisfy D. Observe that, since D extends $D_{\mathcal{V}}$, all the named elements are among $d_1, \ldots, d_j$. For a given $d \notin \{d_1, \ldots, d_j\}$, let $B(d)$ denote the event that $d_1, \ldots, d_j, d$ satisfies D', conditioned on $\psi' \wedge \mathcal{V}$. The probability of $B(d)$, given that $d_1, \ldots, d_j$ satisfies D, is typically very small but is bounded away from 0 by some β independent of N. To see this, note that D' is consistent with $\psi \wedge \mathcal{V}$ (because D' is part of an extension axiom) and so there is a consistent way of choosing how d is related to $d_1, \ldots, d_j$ so as to satisfy D'. Then observe that the total number of possible ways to choose d's properties (as they relate to $d_1, \ldots, d_j$) is independent of N. Since D extends $D_{\mathcal{V}}$, the model fragment defined over the elements $d_1, \ldots, d_j$ satisfies $\psi' \wedge \mathcal{V}$. (Note that it does not necessarily satisfy ψ, which is why we replaced ψ with ψ'.) Since the properties of an element d and its relation to $d_1, \ldots, d_j$ can be chosen independently of the properties of a different element d', the different events $B(d)$, $B(d'), \ldots$ are all independent. Therefore, the probability that there is no domain element at all that, together with $d_1, \ldots, d_j$, satisfies D' is at most $(1-\beta)^{N-j}$. This bounds the probability of the extension axiom being false, relative to fixed $d_1, \ldots, d_j$. There are exactly $\binom{N}{j}$ ways of choosing j elements, so the probability of the axiom being false anywhere in a model is at most $\binom{N}{j}(1-\beta)^{N-j}$. However, this tends to 0 as N goes to infinity. Therefore, the axiom $\forall x_1, \ldots, x_j\, (D \Rightarrow \exists x_{j+1}\, D')$ has asymptotic probability 1 given $\psi' \wedge \mathcal{V}$, and therefore also given $\psi \wedge \mathcal{V}$.

It remains to deal only with the case of a general formula $\varphi \in \mathcal{L}(\Phi)$ such that $T \models \varphi$. By the compactness theorem for first-order logic, if $T \models \varphi$ then there is some finite conjunction of assertions $\xi_1, \ldots, \xi_m \in T$ such that $\wedge_{i=1}^m \xi_i \models \varphi$. By the previous case, each such ξ_i has asymptotic probability 1, and therefore so does this finite conjunction. Hence, the asymptotic probability $\Pr^w_\infty(\varphi \mid \psi \wedge \mathcal{V})$ is also 1. □

As outlined above, this concludes the proof of the main theorem of this section, which we now state.

THEOREM 3.27. *For any sentence* $\varphi \in \mathcal{L}(\Phi)$ *and model description* $\psi \wedge \mathcal{V}$, $\Pr^w_\infty(\varphi \mid \psi \wedge \mathcal{V})$ *is either* 0 *or* 1.

Note that if ψ is a simplified atomic description, then there are no named elements in any model of ψ. Therefore, the only model description augmenting ψ is simply ψ itself. Thus Proposition 3.10, which is Līogon'kīĭ's result, is a corollary of the above theorem.

3.4. Computing the relative weights of model descriptions. We now want to compute the relative weights of model descriptions. It will turn out that certain model descriptions are dominated by others, so that their relative weight is 0, while all the dominating model descriptions have equal weight. Thus, the problem of computing the relative weights of model descriptions reduces to identifying the dominating model descriptions. There are two factors that determine which model descriptions dominate. The first, and more significant, is the number of active atoms; the second is the number of named elements. Let $\alpha(\psi)$ denote the number of active atoms according to ψ.

To compute these relative weights of the model descriptions, we must evaluate $\#world^{\Phi}_N(\psi \wedge \mathcal{V})$. The following lemma gives a precise expression for the asymptotic behavior of this function as N grows large.

LEMMA 3.28. *Let ψ be a consistent atomic description of size $M \geq |\mathcal{C}|$ over Ψ, and let $(\psi \wedge \mathcal{V}) \in \mathcal{M}^{\Phi}(\psi)$.*

(a) *If $\alpha(\psi) = 0$ and $N > \nu(\psi)$, then $\#world_N^{\Psi}(\psi) = 0$. In particular, this holds for all $N > 2^{|\mathcal{P}|}M$.*

(b) *If $\alpha(\psi) > 0$, then*

$$\#world_N^{\Phi}(\psi \wedge \mathcal{V}) \sim \binom{N}{n} a^{N-n} 2^{\sum_{i \geq 2} b_i (N^i - n^i)},$$

where $a = \alpha(\psi)$, $n = \nu(\psi)$, and b_i is the number of predicates of arity i in Φ.

Proof. Suppose that $C_\psi = \langle (f_1, g_1), \ldots, (f_{2^{|\mathcal{P}|}}, g_{2^{|\mathcal{P}|}}) \rangle$ is the characteristic of ψ. Let $\mathcal{W}$ be a model of cardinality N, and let N_i be the number of domain elements in $\mathcal{W}$ satisfying atom A_i. In this case, we say that the *profile of* $\mathcal{W}$ is $\langle N_1, \ldots, N_{2^{|\mathcal{P}|}} \rangle$. Clearly we must have $N_1 + \cdots + N_{2^{|\mathcal{P}|}} = N$. We say that the profile $\langle N_1, \ldots, N_{2^{|\mathcal{P}|}} \rangle$ is *consistent* with C_ψ if $f_i \neq *$ implies that $N_i = f_i$, while $f_i = *$ implies that $N_i \geq M$. Notice that if $\mathcal{W}$ is a model of ψ, then the profile of $\mathcal{W}$ must be consistent with C_ψ.

For part (a), observe that if $\alpha(\psi) = 0$ and $N > \sum_i f_i$, then there can be no models of cardinality N whose profile is consistent with C_ψ. However, if $\alpha(\psi) = 0$, then $\sum_i f_i$ is precisely $\nu(\psi)$. Hence there can be no models of ψ of cardinality N if $N > \nu(\psi)$. Moreover, since $\nu(\psi) \leq 2^{|\mathcal{P}|}M$, the result holds for any $N > 2^{|\mathcal{P}|}$. This proves part (a).

For part (b), let us first consider how many ways there are of choosing a world satisfying $\psi \wedge \mathcal{V}$ with cardinality N and profile $\langle N_1, \ldots, N_{2^{|\mathcal{P}|}} \rangle$. To do the count, we first choose which elements are to be the named elements in the domain. Clearly, there are $\binom{N}{n}$ ways in which this can be done. Once we choose the named elements, their properties are completely determined by $\mathcal{V}$.

It remains to specify the rest of the properties of the world. Let R be a nonunary predicate of arity $i \geq 2$. To completely describe the behavior of R in a world, we need to specify which of the N^i i-tuples over the domain are in the denotation of R. We have already specified this for those i-tuples all of whose components are named elements. There are n^i such i-tuples. Therefore, we have $N^i - n^i$ i-tuples left to specify. Since each subset is a possible denotation, we have $2^{N^i - n^i}$ possibilities for the denotation of R. The overall number of choices for the denotations of all nonunary predicates in the vocabulary is therefore $2^{\sum_{i \geq 2} b_i (N^i - n^i)}$.

It remains only to choose the denotations of the unary predicates for the $N' = N - n$ domain elements that are not named. Let $i_1, \ldots, i_a$ be the active atoms in ψ, and let $h_j = N_{i_j} - g_{i_j}$ for $j = 1, \ldots, a$. Thus, we need to compute all the ways of partitioning the remaining N' elements so that there are h_j elements satisfying atom A_{i_j}; there are $\binom{N'}{h_1\ h_2\ \ldots\ h_a}$ ways of doing this.

We now need to sum over all possible profiles, i.e., those consistent with $\psi \wedge \mathcal{V}$. If $i_j \in A(\psi)$, then there must be at least M domain elements satisfying A_{i_j}. Therefore $N_{i_j} \geq M$, and $h_j = N_{i_j} - g_{i_j} \geq M - g_{i_j}$. This is the only constraint on h_j. Thus, it follows that

$$\#world_N^{\Phi}(\psi \wedge \mathcal{V}) \sim \sum_{\{h_1, \ldots, h_a:\ h_1 + \cdots + h_a = N',\ \forall j\ h_j \geq M - g_{i_j}\}} \binom{N}{n} 2^{\sum_{i \geq 2} b_i (N^i - n^i)} \binom{N'}{h_1\ \ldots\ h_a}.$$

This is equal to

$$\binom{N}{n} 2^{\sum_{i \geq 2} b_i (N^i - n^i)} S$$

for

$$S = \sum_{\{h_1, \ldots, h_a:\ h_1 + \cdots + h_a = N',\ \forall j\ h_j \geq M - g_{i_j}\}} \binom{N'}{h_1\ \ldots\ h_a}.$$

It remains to get a good asymptotic estimate for S. Notice that

$$\sum_{\{h_1,\ldots,h_a:\ h_1+\cdots+h_a=N'\}} \binom{N'}{h_1\ \ldots\ h_a} = a^{N'},$$

since the sum can be viewed as describing all possible ways to assign one of a possible atoms to each of N' elements. Our goal is to show that $a^{N'}$ is actually a good approximation for S as well. Clearly $S < a^{N'}$. Let

$$S_j = \sum_{\{h_1,\ldots,h_a:\ h_j<M,\, h_1+\cdots+h_a=N'\}} \binom{N'}{h_1\ \ldots\ h_a}.$$

Straightforward computation shows that

$$\begin{aligned} S_1 &= \sum_{\{h_1,\ldots,h_a:\ h_1<M,\ h_1+\cdots+h_a=N'\}} \binom{N'}{h_1\ \ldots\ h_a} \\ &= \sum_{h_1=0}^{M-1} \sum_{\{h_2,\ldots,h_a:\ h_2+\cdots+h_a=N'-h_1\}} \binom{N'}{h_1}\binom{N'-h_1}{h_2\ \ldots\ h_a} \\ &\le \sum_{h_1=0}^{M-1} \frac{(N')^{h_1}}{h_1!}(a-1)^{N'-h_1} \\ &< MN^M(a-1)^{N'}. \end{aligned}$$

Similar arguments show that $S_j < MN^M(a-1)^{N'}$ for all j. It follows that

$$\begin{aligned} S &> \sum_{\{h_1,\ldots,h_a:h_1+\cdots+h_a=N'\}} \binom{N'}{h_1\ \ldots\ h_a} - (S_1+\cdots+S_a) \\ &> a^{N'} - aMN^M(a-1)^{N'}. \end{aligned}$$

Therefore,

$$S \sim a^{N'} = a^{N-n},$$

thus concluding the proof. □

The asymptotic behavior described in this lemma motivates the following definition.

DEFINITION 3.29. Given an atomic description ψ over Ψ, let the *degree* of ψ, written $\Delta(\psi)$, be the pair $(\alpha(\psi), \nu(\psi))$, and let degrees be ordered lexicographically. We extend this definition to sentences as follows. For $\theta \in \mathcal{L}(\Psi)$, we define the *degree of* θ *over* Ψ, written $\Delta^{\Psi}(\theta)$, to be $\max_{\psi\in\mathcal{A}_\theta^\Psi} \Delta(\psi)$, and the *activity count* of θ to be $\alpha^{\Psi}(\theta)$ (i.e., the first component of $\Delta^{\Psi}(\theta)$). □

One important conclusion of this lemma justifies our treatment of well-definedness (Definition 2.2) when conditioning on unary formulas. It shows that if θ is satisfied in some "sufficiently large" model, then it is satisfiable over all "sufficiently large" domains.

LEMMA 3.30. *Suppose that* $\theta \in \mathcal{L}(\Psi)$, *and* $M = d(\theta) + |\mathcal{C}_\theta|$. *Then the following conditions are equivalent:*

(a) *θ is satisfied in some model of cardinality greater than* $2^{|\mathcal{P}|}M$,

(b) $\alpha^{\Psi}(\theta) > 0$,

(c) *for all* $N > 2^{|\mathcal{P}|}M$, *θ is satisfiable in some model of cardinality* N,

(d) $\Pr_\infty^w(* \mid \theta)$ *is well defined.*

Proof. By definition, θ is satisfiable in some model of cardinality N iff $\#world_N^{\Psi}(\theta) > 0$. We first show that (a) implies (b). If θ is satisfied in some model of cardinality N greater than $2^{|\mathcal{P}|}M$, then there is some atomic description $\psi \in \mathcal{A}_\theta^{\Psi}$ such that ψ is satisfied in some model of cardinality N. Using part (a) of Lemma 3.28, we deduce that $\alpha(\psi) > 0$ and therefore that $\alpha^{\Psi}(\theta) > 0$. That (b) entails (c) can be verified by examining the proof of Lemma 3.28. That (c) implies (d) and (d) implies (a) is immediate from the definition of well-definedness. □

For the case of sentences in the languages without equality or constants, the condition for well-definedness simplifies considerably.

COROLLARY 3.31. *If* $\theta \in \mathcal{L}^{-}(\mathcal{P})$, *then* $\Pr_\infty^w(* \mid \theta)$ *is well defined iff* θ *is satisfiable.*

Proof. The only if direction is obvious. For the other, if θ is consistent, then it is equivalent to a nonempty disjunction of consistent simplified atomic descriptions. Any consistent simplified atomic description has arbitrarily large models. □

We remark that we can extend our proof techniques to show that Corollary 3.31 holds even if $\mathcal{C} \neq \emptyset$, although we must still require that θ does not mention equality. We omit details here.

For the remainder of this paper, we will consider only sentences θ such that $\alpha^{\Psi}(\theta) > 0$.

Lemma 3.28 shows that, asymptotically, the number of worlds satisfying $\psi \wedge \mathcal{V}$ is completely determined by the degree of ψ. Model descriptions of higher degree have many more worlds, and therefore dominate. On the other hand, model descriptions with the same degree have the same number of worlds at the limit, and are therefore equally likely. This observation allows us to compute the relative weights of different model descriptions.

DEFINITION 3.32. For any degree $\delta = (a, n)$, let $\mathcal{A}_\theta^{\Psi,\delta}$ be the set of atomic descriptions $\psi \in \mathcal{A}_\theta^{\Psi}$ such that $\Delta(\psi) = \delta$. For any set of atomic descriptions $\mathcal{A}'$, we use $\mathcal{M}(\mathcal{A}')$ to denote $\cup_{\psi \in \mathcal{A}'} \mathcal{M}(\psi)$. □

THEOREM 3.33. *Let* $\theta \in \mathcal{L}(\Psi)$ *and* $\Delta^{\Psi}(\theta) = \delta \geq (1, 0)$. *Let* ψ *be an atomic description in* $\mathcal{A}_\theta^{\Psi}$, *and let* $\psi \wedge \mathcal{V} \in \mathcal{M}^{\Phi}(\psi)$.

(a) *If* $\Delta(\psi) < \delta$ *then* $\Pr_\infty^w(\psi \wedge \mathcal{V} \mid \theta) = 0$.

(b) *If* $\Delta(\psi) = \delta$ *then* $\Pr_\infty^w(\psi \wedge \mathcal{V} \mid \theta) = 1/|\mathcal{M}^{\Phi}(\mathcal{A}_\theta^{\Psi,\delta})|$.

Proof. We begin with part (a). Since $\Delta^{\Psi}(\theta) = \delta = (a, n)$, there must exist some atomic description $\psi' \in \mathcal{A}_\theta^{\Psi}$ with $\Delta(\psi') = \delta$. Let $\psi' \wedge \mathcal{V}'$ be some model description in $\mathcal{M}(\psi')$.

$$\begin{aligned}
\Pr_N^w(\psi \wedge \mathcal{V} \mid \theta) &= \frac{\#world_N^{\Phi}(\psi \wedge \mathcal{V})}{\#world_N^{\Phi}(\theta)} \\
&\leq \frac{\#world_N^{\Phi}(\psi \wedge \mathcal{V})}{\#world_N^{\Phi}(\psi' \wedge \mathcal{V}')} \\
&\sim \frac{\binom{N}{\nu(\psi)}(\alpha(\psi))^{N-\nu(\psi)} 2^{\sum_{i \geq 2} b_i (N^i - \nu(\psi)^i)}}{\binom{N}{n} a^{N-n} 2^{\sum_{i \geq 2} b_i (N^i - n^i)}} \\
&= O(N^{\nu(\psi)-n}(\alpha(\psi)/a)^N).
\end{aligned}$$

The last step uses the fact that n and $\nu(\psi)$ can be considered to be constant, and that for any constant k, $\binom{N}{k} \sim N^k/k!$. Since $\Delta(\psi) < \delta = (a, n)$, either $\alpha(\psi) < a$ or $\alpha(\psi) = a$ and $\nu(\psi) < n$. In either case, it is easy to see that $N^{\nu(\psi)-n}(\alpha(\psi)/a)^N$ tends to 0 as $N \rightarrow \infty$, giving us our result.

To prove part (b), we first observe that, due to part (a), we can essentially ignore all model descriptions of low degree. That is:

$$\#world_N^{\Phi}(\theta) \sim \sum_{(\psi' \wedge \mathcal{V}') \in \mathcal{M}(\mathcal{A}_\theta^{\Psi,\delta})} \#world_N^{\Phi}(\psi' \wedge \mathcal{V}').$$

Therefore,

$$\begin{aligned}\Pr^w_N(\psi \wedge \mathcal{V} \mid \theta) &= \frac{\#world^\Phi_N(\psi \wedge \mathcal{V})}{\sum_{(\psi' \wedge \mathcal{V}') \in \mathcal{M}(\mathcal{A}^{\Psi,\delta}_\theta)} \#world^\Phi_N(\psi' \wedge \mathcal{V}')} \\ &\sim \frac{\binom{N}{n} a^{N-n} 2^{\sum_{i\geq 2} b_i(N^i - n^i)}}{\sum_{(\psi' \wedge \mathcal{V}') \in \mathcal{M}(\mathcal{A}^{\Psi,\delta}_\theta)} \binom{N}{n} a^{N-n} 2^{\sum_{i\geq 2} b_i(N^i - n^i)}} \\ &= \frac{1}{|\mathcal{M}(\mathcal{A}^{\Psi,\delta}_\theta)|},\end{aligned}$$

as desired. □

Combining this result with Proposition 3.18, we deduce the following.

THEOREM 3.34. *For any $\varphi \in \mathcal{L}(\Phi)$ and $\theta \in \mathcal{L}(\Psi)$ such that $\Delta^\Psi(\theta) = \delta \geq (1, 0)$,*

$$\Pr^w_\infty(\varphi \mid \theta) = \sum_{(\psi \wedge \mathcal{V}) \in \mathcal{M}(\mathcal{A}^{\Psi,\delta}_\theta)} \Pr^w_\infty(\varphi \mid \psi \wedge \mathcal{V}) / |\mathcal{M}(\mathcal{A}^{\Psi,\delta}_\theta)|.$$

This result, together with the techniques of the next section, will allow us to compute asymptotic conditional probabilities.

The results of Līogon'kīĭ are a simple corollary of the above theorem. For an activity count a, let $\mathcal{A}^{\Psi,a}_\theta$ denote the set of atomic descriptions $\psi \in \mathcal{A}^\Psi_\theta$ such that $\alpha(\psi) = a$.

THEOREM 3.35. [31] *Assume that $\mathcal{C} = \emptyset$, $\varphi \in \mathcal{L}(\Phi)$, $\theta \in \mathcal{L}^-(\mathcal{P})$, and $\alpha^\mathcal{P}(\theta) = a > 0$. Then* $\Pr^w_\infty(\varphi \mid \theta) = \sum_{\psi \in \mathcal{A}^{\mathcal{P},a}_\theta} \Pr^w_\infty(\varphi \mid \psi) / |\mathcal{A}^{\mathcal{P},a}_\theta|$.

Proof. By Theorem 3.9, a sentence $\theta \in \mathcal{L}^-(\mathcal{P})$ is the disjunction of the simplified atomic descriptions in $\mathcal{A}^\mathcal{P}_\theta$. A simplified atomic description ψ has no named elements, and therefore $\Delta(\psi) = (\alpha(\psi), 0)$. Moreover, $\mathcal{M}(\psi) = \{\psi\}$ for any $\psi \in \mathcal{A}^\mathcal{P}_\theta$. The result now follows trivially from the previous theorem. □

This calculation simplifies somewhat if φ and θ are both monadic. In this case, we assume without loss of generality that $d(\varphi) = d(\theta)$. (If not, we can replace φ with $\varphi \wedge \theta$ and θ with $\theta \wedge (\varphi \vee \neg\varphi)$.) This allows us to assume that $\mathcal{A}^\Psi_{\varphi\wedge\theta} \subseteq \mathcal{A}^\Psi_\theta$, thus simplifying the presentation.

COROLLARY 3.36. *Assume that $\varphi, \theta \in \mathcal{L}^-(\mathcal{P})$, and $\alpha^\mathcal{P}(\theta) = a > 0$. Then*

$$\Pr^w_\infty(\varphi \mid \theta) = \frac{|\mathcal{A}^{\mathcal{P},a}_{\varphi\wedge\theta}|}{|\mathcal{A}^{\mathcal{P},a}_\theta|}.$$

Proof. Since φ is monadic, $\varphi \wedge \theta$ is equivalent to a disjunction of the atomic descriptions $\mathcal{A}^\mathcal{P}_{\varphi\wedge\theta} \subseteq \mathcal{A}^\mathcal{P}_\theta$. Atomic descriptions are mutually exclusive; thus, for $\psi \in \mathcal{A}^\Psi_\theta$, $\Pr^w_\infty(\varphi \mid \psi) = 1$ if $\psi \in \mathcal{A}^\Psi_{\varphi\wedge\theta}$ and $\Pr^w_\infty(\varphi \mid \psi) = 0$ otherwise. The result then follows immediately from Theorem 3.35. □

3.5. Asymptotic probabilities for random structures. We now turn our attention to computing asymptotic conditional probabilities using the random-structures method. There are two cases. In the first, there is at least one nonunary predicate in the vocabulary. In this case, random structures is equivalent to random worlds, so that the results in the previous section apply without change.

THEOREM 3.37. *If $\Phi \neq \Psi$ then for any $\varphi \in \mathcal{L}(\Phi)$ and $\theta \in \mathcal{L}(\Psi)$,* $\Pr^{s,\Phi}_\infty(\varphi \mid \theta) = \Pr^w_\infty(\varphi \mid \theta)$.

Proof. Since $\Phi \neq \Psi$, there is at least one nonunary predicate in Φ that does not appear in θ. We can therefore apply Corollary 2.10, and conclude the desired result. □

Random worlds and random structures differ in the second case, when all predicates are unary, but the absence of high-arity predicates makes this a much simpler problem. For the rest of this section, we investigate the asymptotic probability of φ given θ using random structures, for $\varphi, \theta \in \mathcal{L}(\Psi)$. As discussed earlier, we can assume without loss of generality that $\mathcal{A}^{\Psi}_{\varphi\wedge\theta} \subseteq \mathcal{A}^{\Psi}_{\theta}$.

We will use the same basic technique of dividing the structures satisfying θ into classes, and computing the probability of φ on each part. In the case of random structures, however, we partition structures according to the atomic description they satisfy. That is, our computation makes use of the equation

$$\Pr^{s,\Psi}_{\infty}(\varphi \mid \theta) = \sum_{\psi \in \mathcal{A}^{\Psi}_{\theta}} \Pr^{s,\Psi}_{\infty}(\varphi \mid \psi)\Pr^{s,\Psi}_{\infty}(\psi \mid \theta).$$

As for the case of random worlds, we assign weights to atomic descriptions by counting structures. The following lemma computes $\#struct^{\Psi}_{N}(\psi)$ for an atomic description ψ. In the case of random worlds, we saw in Lemma 3.28 that certain model descriptions $\psi \wedge \mathcal{V}$ dominate others, based on the activity count $\alpha(\psi)$ and the number of named elements $\nu(\psi)$ of the atomic description. The following analogue of Lemma 3.28 shows that, for the random-structures method, atomic descriptions of higher activity count $\alpha(\psi)$ dominate regardless of the number of named elements.

LEMMA 3.38. *Let ψ be a consistent atomic description of size $M \geq |\mathcal{C}|$ over Ψ.*

(a) *If $\alpha(\psi) = 0$ and $N > \nu(\psi)$, then $\#struct^{\Psi}_{N}(\psi) = 0$. In particular, this holds for $N > 2^{|\mathcal{P}|}M$.*

(b) *If $\alpha(\psi) > 0$ then $\#struct^{\Psi}_{N}(\psi) \sim \frac{N^{\alpha(\psi)-1}}{(\alpha(\psi)-1)!}$.*

Proof. Part (a) follows immediately from Lemma 3.28(a), since $\#struct^{\Psi}_{N}(\psi) = 0$ iff $\#world^{\Psi}_{N}(\psi) = 0$.

We now proceed to show part (b). Suppose that $C_{\psi} = \langle (f_1, g_1), \ldots, (f_{2^{|\mathcal{P}|}}, g_{2^{|\mathcal{P}|}})\rangle$ is the characteristic of ψ. Let $\mathcal{S}$ be a structure of cardinality N. For any of the models in $\mathcal{S}$, let N_i be the number of domain elements satisfying atom A_i (because $\mathcal{S}$ is an isomorphism class, N_i must be the same for all worlds in the class). As before, we say that the *profile of $\mathcal{S}$* is $\langle N_1, \ldots, N_{2^{|\mathcal{P}|}}\rangle$. Clearly we must have $N_1 + \cdots + N_{2^{|\mathcal{P}|}} = N$. Recall that the profile $\langle N_1, \ldots, N_{2^{|\mathcal{P}|}}\rangle$ is consistent with C_{ψ} if $f_i \neq *$ implies that $N_i = f_i$, while $f_i = *$ implies that $N_i \geq M$. Notice that if $\mathcal{S}$ is a structure of ψ, then the profile of $\mathcal{S}$ must be consistent with C_{ψ}. In fact, there is a unique structure consistent with ψ with cardinality N and profile $\langle N_1, \ldots, N_{2^{|\mathcal{P}|}}\rangle$. This is because a structure is determined by the number of elements in each atom, the assignment of constants to atoms, and the equality relations between the constants. The first part is determined by the profile, while the second and third are determined by ψ. It therefore remains to count only the number of profiles consistent with C_{ψ}. Let $N' = N - \sum_{i \notin A(\psi)} f_i$, and let $i_1, \ldots, i_a$, $a = \alpha(\psi)$, be the active components of C_{ψ}. We want to compute

$$S = |\{\langle N_{i_1}, \ldots, N_{i_a}\rangle : N_{i_1} + \cdots + N_{i_a} = N',\ \forall j\ N_{i_j} \geq M\}| .$$

Notice that, since $\sum_{i \notin A(\psi)} f_i$ and a are constants,

$$|\{\langle N_{i_1}, \ldots, N_{i_a}\rangle : N_{i_1} + \cdots + N_{i_a} = N'\}| = \binom{N'+a-1}{a-1} \sim \frac{(N')^{a-1}}{(a-1)!} \sim \frac{N^{a-1}}{(a-1)!} .$$

As in the proof of Lemma 3.28, let $S_j = |\{\langle N_{i_1}, \ldots, N_{i_a}\rangle : N_{i_1} + \cdots + N_{i_a} = N',\ N_{i_j} < M\}|$. It is easy to see that

$$\begin{aligned} S_1 &= \sum_{N_{i_1}=0}^{M-1} |\{\langle N_{i_2}, \ldots, N_{i_a}\rangle : N_{i_2} + \cdots + N_{i_a} = N' - N_{i_1}\}| \\ &< M(N')^{a-2} , \end{aligned}$$

and similarly for all other S_j. Therefore,

$$\begin{aligned} S &\geq \binom{N'+a-1}{a-1} - (S_1 + \cdots + S_a) \\ &> \binom{N'+a-1}{a-1} - aM(N')^{a-2} \\ &\sim \frac{(N')^{a-1}}{(a-1)!} - aM(N')^{a-2} \\ &\sim \frac{N^{a-1}}{(a-1)!} . \end{aligned}$$

It follows that $S \sim \frac{N^{a-1}}{(a-1)!}$, as desired. □

COROLLARY 3.39. *If $\theta \in \mathcal{L}(\Psi)$, $\alpha^{\Psi}(\theta) = a > 0$, and $\psi \in \mathcal{A}_{\theta}^{\Psi}$, then*

(a) *if $\alpha(\psi) < a$ then $\Pr_{\infty}^{s,\Psi}(\psi \mid \theta) = 0$,*

(b) *if $\alpha(\psi) = a$ then $\Pr_{\infty}^{s,\Psi}(\psi \mid \theta) = 1/|\mathcal{A}_{\theta}^{\Psi,a}|$.*

Proof. Using Lemma 3.38, we can deduce that

$$\Pr_N^{s,\Psi}(\psi \mid \theta) \sim \frac{N^{\alpha(\psi)-1}/(\alpha(\psi)-1)!}{\sum_{\psi' \in \mathcal{A}_{\theta}^{\Psi}} N^{\alpha(\psi')-1}/(\alpha(\psi')-1)!} .$$

As in the proof of Theorem 3.33, we can deduce that if $\alpha(\psi) < a = \alpha^{\Psi}(\theta)$, then $\Pr_{\infty}^{s,\Psi}(\psi \mid \theta) = 0$. Therefore

$$\#struct_N^{\Psi}(\theta) \sim \sum_{\psi' \in \mathcal{A}_{\theta}^{\Psi,a}} \#struct_N^{\Psi}(\psi').$$

Since $\#struct_N^{\Phi}(\psi')$ is asymptotically the same for all ψ' with the same activity count $\alpha(\psi')$, we deduce that if $\alpha(\psi) = a$, then $\Pr_{\infty}^{s,\Psi}(\psi \mid \theta) = 1/|\mathcal{A}_{\theta}^{\Psi,a}|$. □

We can now complete the computation of the value of $\Pr_{\infty}^{s,\Psi}(\varphi \mid \theta)$ for the case of unary φ, θ.

THEOREM 3.40. *If $\varphi, \theta \in \mathcal{L}(\Psi)$ and $a = \alpha^{\Psi}(\theta) > 0$, then*

$$\Pr_{\infty}^{s,\Psi}(\varphi \mid \theta) = \frac{|\mathcal{A}_{\varphi\wedge\theta}^{\Psi,a}|}{|\mathcal{A}_{\theta}^{\Psi,a}|} .$$

Proof. Recall that

$$\Pr_{\infty}^{s,\Psi}(\varphi \mid \theta) = \sum_{\psi \in \mathcal{A}_{\theta}^{\Psi}} \Pr_{\infty}^{s,\Psi}(\varphi \mid \psi) \Pr_{\infty}^{s,\Psi}(\psi \mid \theta).$$

We have already computed $\Pr_{\infty}^{s,\Psi}(\psi \mid \theta)$. It remains to compute $\Pr_{\infty}^{s,\Psi}(\varphi \mid \psi)$ for an atomic description ψ. Recall that $\varphi \wedge \theta$ is equivalent to a disjunction of the atomic descriptions $\mathcal{A}_{\varphi\wedge\theta}^{\Psi} \subseteq \mathcal{A}_{\theta}^{\Psi}$, and that atomic descriptions are mutually exclusive. Therefore, for $\psi \in \mathcal{A}_{\theta}^{\Psi}$, it is easy to see that $\Pr_{\infty}^{s,\Psi}(\varphi \mid \psi) = 1$ if $\psi \in \mathcal{A}_{\varphi\wedge\theta}^{\Psi}$ and $\Pr_{\infty}^{s,\Psi}(\varphi \mid \psi) = 0$ otherwise. Since $\Pr_{\infty}^{s,\Psi}(\psi \mid \theta)$ is 0 except if $\psi \in \mathcal{A}_{\theta}^{\Psi,a}$, it follows from Corollary 3.39 that

$$\Pr_{\infty}^{s,\Psi}(\varphi \mid \theta) = \frac{|\mathcal{A}_{\varphi\wedge\theta}^{\Psi,a}|}{|\mathcal{A}_{\theta}^{\Psi,a}|},$$

as desired. □

Recall that if $\xi \in \mathcal{L}^{-}(\mathcal{P})$, then $\Delta^{\mathcal{P}}(\xi) = (\alpha^{\mathcal{P}}(\xi), 0)$. Thus, comparing Corollary 3.36 with Theorem 3.40 shows that, for formulas in $\mathcal{L}^{-}(\mathcal{P})$, random worlds and random structures are the same.

COROLLARY 3.41. *If $\varphi, \theta \in \mathcal{L}^{-}(\mathcal{P})$, then for any $\Psi \supseteq \mathcal{P}$, $\Pr^{w}_{\infty}(\varphi \mid \theta) = \Pr^{s,\Psi}_{\infty}(\varphi \mid \theta)$.*

Note that, although in general the asymptotic conditional probability in the case of random structures may depend on the vocabulary, for formulas without constant symbols or equality, it does not.

COROLLARY 3.42. *If $\varphi, \theta \in \mathcal{L}^{-}(\mathcal{P})$, and $\mathcal{P}_{\varphi\wedge\theta} \subseteq \Psi \cap \Psi'$ then $\Pr^{s,\Psi}_{\infty}(\varphi \mid \theta) = \Pr^{s,\Psi'}_{\infty}(\varphi \mid \theta) = \Pr^{w}_{\infty}(\varphi \mid \theta)$.*

4. Complexity analysis. In this section we investigate the computational complexity of problems associated with asymptotic conditional probabilities. In fact, we consider three problems: deciding whether the asymptotic probability is well defined, computing it, and approximating it. As we did in the previous section, we begin with the case of random worlds. As we shall see, the same complexity results also hold for the random-structures case (even though, as we have seen, the actual values being computed can differ between random structures and random worlds). The analysis for the unary case of random structures is given in §4.6.

Our computational approach is based on Theorem 3.34, which tells us that

$$\Pr^{w}_{\infty}(\varphi \mid \theta) = \frac{1}{|\mathcal{M}(\mathcal{A}^{\Psi,\delta}_{\theta})|} \cdot \sum_{(\hat{\psi}\wedge\mathcal{V}) \in \mathcal{M}(\mathcal{A}^{\Psi,\delta}_{\theta})} \Pr^{w}_{\infty}(\varphi \mid \hat{\psi} \wedge \mathcal{V}).$$

The basic structure of the algorithms we give for computing $\Pr^{w}_{\infty}(\varphi \mid \theta)$ is simply to enumerate model descriptions $\hat{\psi} \wedge \mathcal{V}$ and, for those of the maximum degree, compute the conditional probability $\Pr^{w}_{\infty}(\varphi \mid \hat{\psi} \wedge \mathcal{V})$. In §4.1 we show how to compute this latter probability.

The complexity of computing asymptotic probabilities depends on several factors: whether the vocabulary is finite, whether there is a bound on the depth of quantifier nesting, whether equality is used in θ, whether nonunary predicates are used, and whether there is a bound on predicate arities. If we consider a fixed and finite vocabulary there are just two cases: if there is no bound on the depth of quantifier nesting then computing probabilities is PSPACE-complete; otherwise the computation can be done in linear time. The case in which the vocabulary is not fixed, which is the case more typically considered in complexity theory, is more complicated. The problem of computing probabilities is complete for the class #EXP (defined below) if either (a) equality is not used in θ and there is some fixed bound on the arity of predicates that can appear in φ, or (b) all predicates in φ are unary. Weakening these conditions in any way—allowing equality while maintaining any arity bound greater than one, or allowing unbounded arity even without using equality in θ—gives the same complexity as the general case (which is complete for a class we call #TA(EXP, LIN), defined later). All these results for the case of an unbounded vocabulary use formulas with quantifier depth 2. As suggested in the introduction, the complexity of the problem drops in the case of formulas of depth 1. A detailed analysis for this case can be found in [27].

4.1. Computing the 0-1 probabilities. The method we give for computing $\Pr^{w}_{\infty}(\varphi \mid \hat{\psi} \wedge \mathcal{V})$ is an extension of Grandjean's algorithm [20] for computing asymptotic probabilities in the unconditional case. For the purposes of this section, fix a model description $\hat{\psi} \wedge \mathcal{V}$ over Φ. In our proof of the conditional 0-1 law (§3.3), we defined a theory T corresponding to $\psi \wedge \mathcal{V}$. We showed that T is a complete and consistent theory, and that $\varphi \in \mathcal{L}(\Phi)$ has asymptotic probability 1 iff $T \models \varphi$. We therefore need an algorithm that decides whether $T \models \varphi$.

Grandjean's original algorithm decides whether $\Pr^w_\infty(\varphi)$ is 0 or 1 for a sentence φ with no constant symbols. For this case, the theory T consists of all possible extension axioms, rather than just the ones involving model descriptions extending $D_\mathcal{V}$ and consistent with ψ (see Definition 3.24). The algorithm has a recursive structure, which at each stage attempts to decide something more general than whether $T \models \varphi$. It decides whether $T \models D \Rightarrow \xi$, where

- D is a complete description over Φ and the set $\mathcal{X}_j = \{x_1, \dots, x_j\}$ of variables, and
- $\xi \in \mathcal{L}(\Phi)$ is a formula whose only free variables (if any) are in $\mathcal{X}_j$.

The algorithm begins with $j = 0$. In this case, D is a complete description over $\mathcal{X}_0$ and Φ. Since Φ contains no constants and $\mathcal{X}_0$ is the empty set, D must in fact be the empty conjunction, which is equivalent to the formula *true*. Thus, for $j = 0$, $T \models D \Rightarrow \varphi$ iff $T \models \varphi$. While $j = 0$ is the case of real interest, the recursive construction Grandjean uses forces us to deal with the case $j > 0$ as well. In this case, the formula $D \Rightarrow \varphi$ contains free variables; these variables are treated as being universally quantified for purposes of determining if $T \models D \Rightarrow \varphi$.

Our algorithm is the natural extension to Grandjean's algorithm for the case of conditional probabilities and for a language with constants. The chief difference is that we begin by considering $T \models D_\mathcal{V} \Rightarrow \varphi$ (where $\mathcal{V}$ is the model fragment on which we are conditioning). Suppose $D_\mathcal{V}$ uses the variables $x_1, \dots, x_n$, where $n = \nu(\psi)$. We have said that $T \models D_\mathcal{V} \Rightarrow \varphi$ is interpreted as $T \models \forall x_1, \dots, x_n\ (D_\mathcal{V} \Rightarrow \varphi)$, and this is equivalent to $T \models (\exists x_1, \dots, x_n\ D_\mathcal{V}) \Rightarrow \varphi$ because φ is closed. Because $\exists x_1, \dots, x_n\ D_\mathcal{V}$ is in T by definition, this latter assertion is equivalent to $T \models \varphi$, which is what we are really interested in.

Starting from the initial step just outlined, the algorithm then recursively examines smaller and smaller subformulas of φ, while maintaining a description D which keeps track of any new free variables that appear in the current subformula. Of course, D will also extend $D_\mathcal{V}$ and will be consistent with ψ.

We now describe the algorithm in more detail. Without loss of generality, we assume that all negations in φ are pushed in as far as possible, so that only atomic formulas are negated. We also assume that φ does not use the variables $x_1, x_2, x_3, \dots$. The algorithm proceeds by induction on the structure of the formula, until the base case—an atomic formula or its negation—is reached. The following equivalences form the basis for the recursive procedure:

1. If ξ is of the form ξ' or $\neg\xi'$ for an atomic formula ξ', then $T \models D \Rightarrow \xi$ iff ξ is a conjunct of D.
2. If ξ is of the form $\xi_1 \wedge \xi_2$, then $T \models D \Rightarrow \xi$ iff $T \models D \Rightarrow \xi_1$ and $T \models D \Rightarrow \xi_2$.
3. If ξ is of the form $\xi_1 \vee \xi_2$ then $T \models D \Rightarrow \xi$ iff $T \models D \Rightarrow \xi_1$ or $T \models D \Rightarrow \xi_2$.
4. If ξ is of the form $\exists y\, \xi'$ and D is a complete description over Φ and $\{x_1, \dots, x_j\}$, then $T \models D \Rightarrow \xi$ iff $T \models D' \Rightarrow \xi'[y/x_{j+1}]$ for some complete description D' over Φ and $\{x_1, \dots, x_{j+1}\}$ that extends D and is consistent with ψ.
5. If ξ is of the form $\forall y\, \xi'$ and D is a complete description over Φ and $\{x_1, \dots, x_j\}$, then $T \models D \Rightarrow \xi$ iff $T \models D' \Rightarrow \xi'[y/x_{j+1}]$ for all complete descriptions D' over Φ and $\{x_1, \dots, x_{j+1}\}$ that extend D and are consistent with ψ.

The proof that this procedure is correct is based on the following proposition, which can easily be proved using the same techniques as for Proposition 3.25.

PROPOSITION 4.1. *If D is a complete description over Φ and $\mathcal{X}$ and $\xi \in \mathcal{L}(\Phi)$ is a formula all of whose free variables are in $\mathcal{X}$, then either $T \models D \Rightarrow \xi$ or $T \models D \Rightarrow \neg\xi$.*

Proof. We know that T has no finite models. By the Löwenheim–Skolem Theorem [12, p. 141], we can, without loss of generality, restrict attention to countably infinite models of T.

Suppose $\mathcal{X} = \{x_1, x_2, \ldots, x_j\}$ and that $T \not\models D \Rightarrow \xi$. Then there is some countable model $\mathcal{U}$ of T, and j domain elements $\{d_1, \ldots, d_j\}$ in the domain of $\mathcal{U}$, which satisfy $D \wedge \neg\xi$. Consider another model $\mathcal{U}'$ of T, and any $\{d'_1, \ldots, d'_j\}$ in the domain of U' that satisfy D. Because D is a complete description, the substructures over $\{d_1, \ldots, d_j\}$ and $\{d'_1, \ldots, d'_j\}$ are isomorphic. We can use the back and forth construction of Proposition 3.25 to extend this to an isomorphism between $\mathcal{U}$ and $\mathcal{U}'$. But then it follows that $\{d'_1, \ldots, d'_j\}$ must also satisfy $\neg\xi$. Since $\mathcal{U}$ was arbitrary, $T \models D \Rightarrow \neg\xi$. The result follows. □

The following result shows that the algorithm above gives a sound and complete procedure for determining whether $T \models D_{\mathcal{V}} \Rightarrow \varphi$.

THEOREM 4.2. *Each of the equivalences in steps* (1)–(5) *above is true.*

Proof. The equivalences for steps (1)–(3) are easy to show, using Proposition 4.1. To prove (4), consider some formula $D \Rightarrow \exists y\, \xi'$, where D is a complete description over $x_1, \ldots, x_j$ and the free variables of ξ are contained in $\{x_1, \ldots, x_j\}$. Let $\mathcal{U}$ be some countable model of T, and let $d_1, \ldots, d_j$ be elements in $\mathcal{U}$ that satisfy D. If $\mathcal{U}$ satisfies $D \Rightarrow \exists y\, \xi'$ then there must exist some other element d_{j+1} that, together with $d_1, \ldots, d_j$, satisfies ξ. Consider the description D' over $x_1, \ldots, x_{j+1}$ that extends D and is satisfied by $d_1, \ldots, d_{j+1}$. Clearly $T \not\models D' \Rightarrow \neg\xi'[y/x_{j+1}]$ because this is false in $\mathcal{U}$. So, by Proposition 4.1, $T \models D' \Rightarrow \xi'[y/x_{j+1}]$ as required.

For the other direction, suppose that $T \models D' \Rightarrow \xi'[y/x_{j+1}]$ for some D' extending D. It follows that $T \models \exists x_{j+1} D' \Rightarrow \exists x_{j+1}\xi'[y/x_{j+1}]$. The result follows from the observation that T contains the extension axiom $\forall x_1, \ldots, x_j(D \Rightarrow \exists x_{j+1} D')$.

The proof for case (5) is similar to that for case (4), and is omitted. □

We analyze the complexity of this algorithm in terms of alternating Turing machines (ATMs) [5]. Recall that in an ATM, the nonterminal states are classified into two kinds: universal and existential. Just as with a nondeterministic TM, a nonterminal state may have one or more successors. The terminal states are classified into two kinds: accepting and rejecting. The computation of an ATM forms a tree, where the nodes are instantaneous descriptions (IDs) of the machine's state at various points in the computation, and the children of a node are the possible successor IDs. We recursively define what it means for a node in a computation tree to be an *accepting* node. Leaves are terminal states, and a leaf is accepting just if the machine is in an accepting state in the corresponding ID. A node whose ID is in an existential state is accepting iff at least one of its children is accepting. A node whose ID is in a universal state is accepting iff all of its children are accepting. The entire computation is accepting if the root is an accepting node.

We use several different measures for the complexity of an ATM computation. The time of the computation is the number of steps taken by its longest computation branch. The number of *alternations* of a computation of an ATM is the maximum number of times, over all branches, that the type of state switched (from universal to existential or vice versa). The number of *branches* is simply the number of distinct computation paths. The number of branches is always bounded by an exponential in the computation time, but sometimes we can find tighter bounds.

Grandjean's algorithm, and our variant of it, is easily implemented on an ATM. Each inductive step corresponding to a disjunction or an existential quantifier can be implemented using a sequence of existential guesses. Similarly, each step corresponding to a conjunction or a universal quantifier can be implemented using a sequence of universal guesses. Note that the number of alternations is at most $|\varphi|$. We must analyze the time and branching complexity of this ATM. Given $\psi \wedge \mathcal{V}$, each computation branch of this ATM can be regarded as doing the following. It

(a) constructs a complete description D over the variables $x_1, \ldots, x_{n+k}$ that extends $D_{\mathcal{V}}$ and is consistent with ψ, where $n = \nu(\psi)$ and $k \leq |\varphi|/2$ is the number of variables appearing in φ,

(b) chooses a formula ξ or $\neg\xi$, where ξ is an atomic subformula of φ (with free variables renamed appropriately so that they are included in $\{x_1, \ldots, x_{n+k}\}$), and

(c) checks whether $T \models D \Rightarrow \xi$.

Generating a complete description D requires time $|D|$, and if we construct D by adding conjuncts to $D_{\mathcal{V}}$ then it is necessarily the case that D extends $D_{\mathcal{V}}$. To check whether D is consistent with ψ, we must verify that D does not assert the existence of any new element in any finite atom. Under an appropriate representation of ψ (outlined after Corollary 4.4 below), this check can be done in time $O(|D|2^{|\mathcal{P}|})$. Choosing an atomic subformula ξ of φ can take time $O(|\varphi|)$. Finally, checking whether $T \models D \Rightarrow \xi$ can be accomplished by simply scanning $|D|$. It is easy to see that we can do this without backtracking over $|D|$. Since $|D| > |\xi|$, it can be done in time $O(|D|)$. Combining all these estimates, we conclude that the length of each branch is $O(|D|2^{|\mathcal{P}|} + |\varphi|)$.

Let D be any complete description over Φ and $\mathcal{X}$. Without loss of generality, we assume that each constant in Φ is equal to (at least) one of the variables in $\mathcal{X}$. To fully describe D we must specify, for each predicate R of arity i, which of the i-tuples of variables used in D satisfy R. Thus, the number of choices needed to specify the denotation of R is bounded by $|\mathcal{X}|^{\rho}$ where ρ is the maximum arity of a predicate in Φ. Therefore, $|D|$ is $O(|\Phi||\mathcal{X}|^{\rho})$. In the case of the description D generated by the algorithm, $\mathcal{X}$ is $\{x_1, \ldots, x_n, x_{n+1}, \ldots, x_{n+k}\}$, and $n + k$ is less than $n + |\varphi|$. Thus, the length of such a description D is $O(|\Phi|(n + |\varphi|)^{\rho})$.

Using this expression, and our analysis above, we see that the computation time is certainly $O(|\Phi|2^{|\mathcal{P}|}(n + |\varphi|)^{\rho})$. In general, the number of branches of the ATM is at most the number of complete descriptions multiplied by the number of atomic formulas in φ. The first of these terms can be exponential in the length of each description. Therefore the number of branches is $O(|\varphi|2^{|\Phi|(n+|\varphi|)^{\rho}}) = 2^{O(|\Phi|(n+|\varphi|)^{\rho})}$. We can, however, get a better bound on the number of branches if all predicates in Φ are unary (i.e., if $\rho = 1$). In this case, ψ already specifies all the properties of the named elements. Therefore, a complete description D is determined when we decide, for each of the at most k variables in D not corresponding to named elements, whether it is equal to a named element and, if not, which atom it satisfies. It follows that there are at most $(2^{|\Phi|} + n)^k$ complete descriptions in this case, and so at most $|\varphi|(2^{|\Phi|} + n)^k$ branches. Since $k \leq |\varphi|/2$, the number of branches is certainly $O((2^{|\Phi|} + n)^{|\varphi|})$ if $\rho = 1$. We summarize this analysis in the following theorem, which forms the basis for almost all of our upper bounds in this section.

THEOREM 4.3. *There exists an alternating Turing machine that takes as input a finite vocabulary Φ, a model description $\psi \wedge \mathcal{V}$ over Φ, and a formula $\varphi \in \mathcal{L}(\Phi)$, and decides whether* $\Pr^w_\infty(\varphi \mid \psi \wedge \mathcal{V})$ *is* 0 *or* 1. *The machine uses time $O(|\Phi|2^{|\mathcal{P}|}(\nu(\psi) + |\varphi|)^{\rho})$ and $O(|\varphi|)$ alternations, where ρ is the maximum arity of predicates in Φ. If $\rho > 1$, the number of branches is $2^{O(|\Phi|(\nu(\psi)+|\varphi|)^{\rho})}$. If $\rho = 1$, the number of branches is $O((2^{|\Phi|} + \nu(\psi))^{|\varphi|})$.*

An alternating Turing machine can be simulated by a deterministic Turing machine which traverses all possible branches of the ATM, while keeping track of the intermediate results necessary to determine whether the ATM accepts or rejects. The time taken by the deterministic simulation is linear in the product of the number of branches of the ATM and the time taken by each branch. The space required is the logarithm of the number of branches plus the space required for each branch. In this case, both these terms are $O(|D| + |\varphi|)$, where D is the description generated by the machine. This allows us to prove the following important corollary.

```
Procedure $Compute\text{-}Pr_\infty(\varphi \mid \theta)$
    $\delta \leftarrow (0, 0)$
    For each model description $\psi \wedge \mathcal{V}$ do:
        Compute $\Pr^w_\infty(\theta \mid \psi \wedge \mathcal{V})$ using our variant of Grandjean's algorithm
        If $\Delta(\psi) = \delta$ and $\Pr^w_\infty(\theta \mid \psi \wedge \mathcal{V}) = 1$ then
            $count(\theta) \leftarrow count(\theta) + 1$
            Compute $\Pr^w_\infty(\varphi \mid \psi \wedge \mathcal{V})$ using our variant of Grandjean's algorithm
            $count(\varphi) \leftarrow count(\varphi) + \Pr^w_\infty(\varphi \mid \psi \wedge \mathcal{V})$
        If $\Delta(\psi) > \delta$ and $\Pr^w_\infty(\theta \mid \psi \wedge \mathcal{V}) = 1$ then
            $\delta \leftarrow \Delta(\psi)$
            $count(\theta) \leftarrow 1$
            Compute $\Pr^w_\infty(\varphi \mid \psi \wedge \mathcal{V})$ using our variant of Grandjean's algorithm
            $count(\varphi) \leftarrow \Pr^w_\infty(\varphi \mid \psi \wedge \mathcal{V})$
    If $\delta = (0, 0)$ then output "$\Pr^w_\infty(\varphi \mid \theta)$ not well defined"
        otherwise output "$\Pr^w_\infty(\varphi \mid \theta) = count(\varphi)/count(\theta)$".
```

FIG. 1. Compute-$\Pr_\infty$ *for computing asymptotic conditional probabilities.*

COROLLARY 4.4. *There exists a deterministic Turing machine that takes as input a finite vocabulary Φ, a model description $\psi \wedge \mathcal{V}$ over Φ, and a formula $\varphi \in \mathcal{L}(\Phi)$, and decides whether $\Pr^w_\infty(\varphi \mid \psi \wedge \mathcal{V})$ is 0 or 1. If $\rho > 1$ the machine uses time $2^{O(|\Phi|(\nu(\psi)+|\varphi|)^\rho)}$ and space $O(|\Phi|(\nu(\psi) + |\varphi|)^\rho)$. If $\rho = 1$ the machine uses time $2^{O(|\varphi||\Phi|\log(\nu(\psi)+1))}$ and space $O(|\varphi||\Phi|\log(\nu(\psi) + 1))$.*

4.2. Computing asymptotic conditional probabilities. Our overall goal is to compute $\Pr^w_\infty(\varphi \mid \theta)$ for some $\varphi \in \mathcal{L}(\Phi)$ and $\theta \in \mathcal{L}(\Psi)$. To do this, we enumerate model descriptions over Φ of size $d(\theta) + |\mathcal{C}|$, and check which are consistent with θ. Among those model descriptions that are of maximal degree, we compute the fraction of model descriptions $\psi \wedge \mathcal{V}$ for which $\Pr^w_\infty(\varphi \mid \psi \wedge \mathcal{V})$ is 1.

More precisely, let $\delta_\theta = \Delta^\Psi(\theta)$. Theorem 3.34 tells us that

$$\Pr^w_\infty(\varphi \mid \theta) = \frac{1}{|\mathcal{M}(\mathcal{A}_\theta^{\Psi,\delta_\theta})|} \sum_{(\psi \wedge \mathcal{V}) \in \mathcal{M}(\mathcal{A}_\theta^{\Psi,\delta_\theta})} \Pr^w_\infty(\varphi \mid \psi \wedge \mathcal{V}).$$

The procedure *Compute-Pr_∞*, described in Fig. 1, generates one by one all model descriptions of size $d(\theta) + |\mathcal{C}|$ over Φ. The algorithm keeps track of three things, among the model descriptions considered thus far: (1) the highest degree δ of a model description consistent with θ, (2) the number $count(\theta)$ of model descriptions of degree δ consistent with θ, and (3) among the model descriptions of degree δ consistent with θ, the number $count(\varphi)$ of descriptions such that $\Pr^w_\infty(\varphi \mid \psi \wedge \mathcal{V}) = 1$. Thus, for each model description $\psi \wedge \mathcal{V}$ generated, the algorithm computes $\Delta(\psi)$. If $\Delta(\psi) < \delta$ or $\Pr^w_\infty(\theta \mid \psi \wedge \mathcal{V})$ is 0, then the model description is ignored. Otherwise, if $\Delta(\psi) > \delta$, then the count for lower degrees is irrelevant. In this case, the algorithm erases the previous counts by setting $\delta \leftarrow \Delta(\psi)$, $count(\theta) \leftarrow 1$, and $count(\varphi) \leftarrow \Pr^w_\infty(\varphi \mid \psi \wedge \mathcal{V})$. If $\Delta(\psi) = \delta$, then the algorithm updates $count(\theta)$ and $count(\varphi)$ appropriately.

Examining *Compute-Pr_∞*, we see that its complexity is dominated by two major quantities: the time required to generate all model descriptions, and the time required to compute each 0-1 probability using our variant of Grandjean's algorithm. The complexity of the latter was given in Theorem 4.3 and Corollary 4.4. The following proposition states the length of a model description; the time required to generate all model descriptions is exponential in this length.

PROPOSITION 4.5. *If $M > |\mathcal{C}|$ then the length of a model description of size M over Φ is*

$$O(|\Phi|(2^{|\mathcal{P}|}M)^{\rho}).$$

Proof. Consider a model description over Φ of size $M = d(\theta) + |\mathcal{C}|$. Such a model description consists of two parts: an atomic description ψ over Ψ and a model fragment $\mathcal{V}$ over Φ which is in $\mathcal{M}(\psi)$. To specify an atomic description ψ, we need to specify the unary properties of the named elements; furthermore, for each atom, we need to say whether it has any elements beyond the named elements (i.e., whether it is active). Using this representation, the size of an atomic description ψ is $O(|\Psi|\nu(\psi) + 2^{|\mathcal{P}|})$. As we have already observed, the length of a complete description D over Φ and $\mathcal{X}$ is $O(|\Phi||\mathcal{X}|^{\rho})$. In the case of a description $D_{\mathcal{V}}$ for $\mathcal{V} \in \mathcal{M}(\psi)$, this is $O(|\Phi|\nu(\psi)^{\rho})$. Using $\nu(\psi) \leq 2^{|\mathcal{P}|}M$, we obtain the desired result. □

Different variants of this algorithm are the basis for most of the upper bounds in the remainder of this section.

4.3. Finite vocabulary. We now consider the complexity of various problems related to $\Pr_{\infty}^{w}(\varphi \mid \theta)$ for a fixed finite vocabulary Φ. The input for such problems is simply φ and θ, and so the input length is the sum of the lengths of φ and θ. Since, for the purposes of this section, we view the vocabulary Φ as fixed (independent of the input), its size and maximum arity can be treated as constants.

We first consider the issue of well-definedness.

THEOREM 4.6. *Fix a finite vocabulary Φ with at least one unary predicate symbol. For $\theta \in \mathcal{L}(\Psi)$, the problem of deciding whether $\Pr_{\infty}^{w}(* \mid \theta)$ is well defined is PSPACE-complete. The lower bound holds even if $\theta \in \mathcal{L}^{-}(\{P\})$.*

Proof. It follows from Lemma 3.30 that $\Pr_{\infty}^{w}(* \mid \theta)$ is well defined iff $\alpha^{\Psi}(\theta) > 0$. This is true iff there is some atomic description $\psi \in \mathcal{A}_{\theta}^{\Psi}$ such that $\alpha(\psi) > 0$. This holds iff there exists an atomic description ψ of size $M = d(\theta) + |\mathcal{C}|$ over Ψ and some model fragment $\mathcal{V} \in \mathcal{M}^{\Psi}(\psi)$ such that $\alpha(\psi) > 0$ and $\Pr_{\infty}^{w}(\theta \mid \psi \wedge \mathcal{V}) = 1$. Since we are working within Ψ, we can take $\rho = 1$ and $|\mathcal{P}|$ to be a constant, independent of θ. Thus, the length of a model description $\psi \wedge \mathcal{V}$ as given in Proposition 4.5 is polynomial in $|\theta|$. It is therefore possible to generate model descriptions in PSPACE. Using Corollary 4.4, we can check, in polynomial space, for a model description $\psi \wedge \mathcal{V}$ whether $\Pr_{\infty}^{w}(\theta \mid \psi \wedge \mathcal{V})$ is 1. Therefore, the entire procedure can be done in polynomial space.

For the lower bound, we use a reduction from the problem of checking the truth of quantified Boolean formulas (QBF), a problem well known to be PSPACE-complete [37]. The reduction is similar to that used to show that checking whether a first-order sentence is true in a given finite structure is PSPACE-hard [6]. Given a quantified Boolean formula β, we define a first-order sentence $\xi_{\beta} \in \mathcal{L}^{-}(\{P\})$ as follows. The structure of ξ_{β} is identical to that of β, except that any reference to a propositional variable x, except in the quantifier, is replaced by $P(x)$. For example, if β is $\forall x \exists y\, (x \wedge y)$, ξ_{β} will be $\forall x \exists y\, (P(x) \wedge P(y))$. Let θ be $\xi_{\beta} \wedge \exists x\, P(x) \wedge \exists x\, \neg P(x)$. Clearly, $\Pr_{\infty}^{w}(* \mid \theta)$ is well defined exactly if β is true. □

In order to compute asymptotic conditional probabilities in this case, we simply use *Compute-Pr*$_{\infty}$. In fact, since *Compute-Pr*$_{\infty}$ can also be used to determine well-definedness, we could also have used it to prove the previous theorem.

THEOREM 4.7. *Fix a finite vocabulary Φ. For $\varphi \in \mathcal{L}(\Phi)$ and $\theta \in \mathcal{L}(\Psi)$, the problem of computing $\Pr_{\infty}^{w}(\varphi \mid \theta)$ is PSPACE-complete. Indeed, deciding if $\Pr_{\infty}^{w}(\varphi \mid true) = 1$ is PSPACE-hard even if $\varphi \in \mathcal{L}^{-}(\{P\})$ for some unary predicate symbol P.*

Proof. The upper bound is obtained directly from *Compute-Pr*$_{\infty}$ in Fig. 1. The algorithm generates model descriptions one by one. Using the assumption that Φ is fixed and finite, each model description has polynomial length, so that this can be done in PSPACE. Corollary 4.4 implies that, for a fixed finite vocabulary, the 0-1 probabilities for each model description can also be computed in polynomial space. While *count*(θ) and *count*(φ) can be exponential (as large as the number of model descriptions), only polynomial space is required for their binary representation. Thus, *Compute-Pr*$_\infty$ works in PSPACE under the assumption of a fixed finite vocabulary.

For the lower bound, we provide a reduction from QBF much like that used in Theorem 4.6. Given a quantified Boolean formula ξ and a unary predicate symbol P, we construct a sentence $\xi_\beta \in \mathcal{L}^-(\{P\})$ just as in the proof of Theorem 4.6. It is easy to see that $\Pr^w_\infty(\xi_\beta \mid \mathit{true}) = 1$ iff β is true. (By the unconditional 0-1 law, $\Pr^w_\infty(\xi_\beta \mid \mathit{true})$ is necessarily either 0 or 1.) □

It follows immediately from Theorem 4.7 that we cannot approximate the limit. Indeed, if we fix ϵ with $0 < \epsilon < 1$, the problem of deciding whether $\Pr^w_\infty(\varphi \mid \theta) \in [0, 1-\epsilon]$ is PSPACE-hard even for $\varphi, \theta \in \mathcal{L}^-(\{P\})$. We might hope to prove that for any nontrivial interval $[r_1, r_2]$, it is PSPACE-hard to decide if $\Pr^w_\infty(\varphi \mid \theta) \in [r_1, r_2]$. This stronger lower bound does not hold for the language $\mathcal{L}^-(\{P\})$. Indeed, it follows from Theorem 3.35 that if Φ is any fixed vocabulary then, for $\varphi \in \mathcal{L}(\Phi)$ and $\theta \in \mathcal{L}^-(\Psi)$, $\Pr^w_\infty(\varphi \mid \theta)$ must take one of a finite number of values (the possible values being determined entirely by Φ). So the approximation problem is frequently trivial; in particular, this is the case for any $[r_1, r_2]$ that does not contain one of the possible values. To see that there are only a finite number of values, first note that there is a fixed collection of atoms over Φ. If θ does not use equality, an atomic description can only say, for each atom A over Φ, whether $\exists x\, A(x)$ or $\neg\exists x\, A(x)$ holds. There is also a fixed set of constant symbols to describe. Therefore, there is a fixed set of possible atomic descriptions. Finally, note that the only named elements are the constants, and so there is also a fixed (and finite) set of model fragments. This shows that the set of model descriptions is finite, from which it follows that $\Pr^w_\infty(\varphi \mid \theta)$ takes one of finitely many values fixed by Φ. Thus, in order to have $\Pr^w_\infty(\varphi \mid \theta)$ assume infinitely many values, we must allow equality in the language. Moreover, even with equality in the language, one unary predicate does not suffice. Using Theorem 3.34, it can be shown that two unary predicates are necessary to allow the asymptotic conditional probability to assume infinitely many possible values. As the following result shows, this condition also suffices.

THEOREM 4.8. *Fix a finite vocabulary Φ that contains at least two unary predicates and rational numbers $0 \le r_1 \le r_2 \le 1$ such that $[r_1, r_2] \ne [0, 1]$. For $\varphi, \theta \in \mathcal{L}(\mathcal{P})$, the problem of deciding whether $\Pr^w_\infty(\varphi \mid \theta) \in [r_1, r_2]$ is PSPACE-hard, even given an oracle that tells us whether the limit is well defined.*

Proof. We first show that, for any rational number r with $0 < r < 1$, we can construct φ_r, θ_r such that $\Pr^w_\infty(\varphi_r \mid \theta_r) = r$. Suppose $r = q/p$. We assume, without loss of generality, that $\Phi = \{P, Q\}$. Let θ_r be the sentence

$$\exists^{p-1}x\, P(x) \wedge \big(\exists^{q-1}x\,(P(x) \wedge Q(x)) \vee \exists^{q}x\,(P(x) \wedge Q(x))\big) \wedge \exists^{0}x\,(\neg P(x) \wedge \neg Q(x)).$$

That is, no elements satisfy the atom $\neg P \wedge \neg Q$, either q or $q-1$ elements satisfy the atom $P \wedge Q$, and $p-1$ elements satisfy P. Thus, there are exactly two atomic descriptions consistent with θ_r. In one of them, ψ_1, there are $q-1$ elements satisfying $P \wedge Q$ and $p-q$ elements satisfying $P \wedge \neg Q$ (all the remaining elements satisfy $\neg P \wedge Q$). In the other, ψ_2, there are q elements satisfying $P \wedge Q$ and $p-q-1$ elements satisfying $P \wedge \neg Q$. Clearly, the degree of ψ_1 is the same as that of ψ_2, so that neither one dominates. In particular, both define $p-1$ named elements. The number of model fragments for ψ_1 is $\binom{p-1}{q-1} = \frac{(p-1)!}{(q-1)!(p-q)!}$. The number of model fragments for ψ_2 is $\binom{p-1}{q} = \frac{(p-1)!}{q!(p-q-1)!}$. Let φ_r be ψ_1. Clearly

$$\begin{aligned}\Pr^w_\infty(\varphi_r \mid \theta_r) &= \frac{|\mathcal{M}(\psi_1)|}{|\mathcal{M}(\psi_1)| + |\mathcal{M}(\psi_2)|} \\ &= \frac{(p-1)!/((q-1)!(p-q)!)}{(p-1)!/((q-1)!(p-q)!) + (p-1)!/(q!(p-q-1)!)} \\ &= \frac{q}{q+(p-q)} = \frac{q}{p} = r\,.\end{aligned}$$

Now, assume we are given $r_1 \leq r_2$. We prove the result by reduction from QBF, as in the proof of Theorem 4.6. If $r_1 = 0$ then the result follows immediately from Theorem 4.7. If $0 < r_1 = q/p$, let β be a QBF, and consider $\Pr^w_\infty(\xi_\beta \wedge \varphi_{r_1} \mid \theta_{r_1} \wedge \exists x\, \neg P(x))$. Note that, since $p \geq 2$, θ_{r_1} implies $\exists x\, P(x)$. It is therefore easy to see that this probability is 0 if β is false and $\Pr^w_\infty(\varphi_{r_1} \mid \theta_{r_1}) = r_1$ otherwise. Thus, we can check if β is true by deciding whether $\Pr^w_\infty(\xi_\beta \wedge \varphi_{r_1} \mid \theta_{r_1} \wedge \exists x\, \neg P(x)) \in [r_1, r_2]$. This proves PSPACE-hardness.[9] □

These results show that simply assuming that the vocabulary is fixed and finite is not by itself enough to lead to computationally easy problems. Nevertheless, there is some good news. We observed in a companion paper [23] that if Φ is fixed and finite, and we bound the depth of quantifier nesting, then there exists a linear time algorithm for computing asymptotic probabilities. In general, as we observed in [23], we cannot effectively construct this algorithm, although we know that it exists. As we now show, for the case of conditioning on a unary formula, we can effectively construct this algorithm.

THEOREM 4.9. *Fix $d \geq 0$. For $\varphi \in \mathcal{L}(\Phi)$, $\theta \in \mathcal{L}(\Psi)$ such that $d(\varphi), d(\theta) \leq d$, we can effectively construct a linear time algorithm that decides if $\Pr^w_\infty(\varphi \mid \theta)$ is well defined and computes it if it is.*

Proof. The proof of the general theorem in [23] shows that if there is a bound d on the quantification depth of formulas and a finite vocabulary, then there is a finite set Σ_d of formulas such that every formula ξ of depth at most d is equivalent to a formula in Σ_d. Moreover, we can construct an algorithm that, given such a formula ξ, will in linear time find some formula equivalent to ξ in Σ_d. (We say "some" rather than "the," because it is necessary for the algorithm's constructibility that there will generally be several formulas equivalent to ξ in Σ_d.) Given this, the problem reduces to constructing a lookup table for the asymptotic conditional probabilities for all formulas in Σ_d. In general, there is no effective technique for constructing this table. However, if we allow conditioning only on unary formulas, it follows from Theorem 4.7 that there is. The result now follows. □

4.4. Infinite vocabulary—restricted cases.

In the next two sections we consider an infinite vocabulary Ω. As discussed in §2.3, there are at least two distinct interpretations for asymptotic conditional probabilities in the case of an infinite vocabulary. One interpretation of "infinite vocabulary" views Ω as a potential or background vocabulary, so that every problem instance includes as part of its input the actual finite subvocabulary that is of interest. So, although this subvocabulary is finite, there is no bound on its possible size. The alternative is to interpret infinite vocabularies more literally, using the limit process explained in §2.3. In the case of the random-worlds method, Proposition 2.1 shows that both interpretations give the same result. Thus, it is immediate that all complexity results we prove with respect to one interpretation immediately hold for the other. As we are postponing the discussion of random structures to §4.6, we present the earlier results with respect to the second, less cumbersome, interpretation.

[9] In this construction, it is important to note that although φ_{r_1} and θ_{r_1} can be long sentences, their length depends only on r_1, which is treated as being fixed. Therefore, the constructed asymptotic probability expression does have length polynomial in $|\beta|$. This is also the case in similar constructions later in the paper.

As before, we are interested in computing the complexity of the same three problems: deciding whether the asymptotic probability is well defined, computing it, and approximating it. As we mentioned earlier, the complexity is quite sensitive to a number of factors. One factor, already observed in the unconditional case [4], [20], is whether there is a bound on the arity of the predicates in Ω. Without such a bound, the problem is complete for the class #TA(EXP,LIN). Unlike the unconditional case, however, simply putting a bound on the arity of the predicates in Ω is not enough to improve the complexity (unless the bound is 1); we also need to restrict the use of equality, so that it cannot appear in the right-hand side of the conditional. Roughly speaking, with equality, we can use the named elements to play the same role as the predicates of unbounded arity. In this section, we consider what happens if we in fact restrict the language so that either (1) Ω has no predicate of arity ≥ 2, or (2) there is a bound (which may be greater than 1) on the arity of the predicates in Ω, but we never condition on formulas that use equality. As we now show, these two cases turn out to be quite similar. In particular, the same complexity results hold.

Throughout this section, we take Ω to be a fixed infinite vocabulary such that all predicate symbols in Ω have arity less than some fixed bound ρ. Let $\mathcal{Q}$ be the set of all unary predicate symbols in Ω, let $\mathcal{D}$ be the set of all constant symbols in Ω, and let $\Upsilon = \mathcal{Q} \cup \mathcal{D}$.

We start with the problem of deciding whether the asymptotic probability is well defined. Since well-definedness depends only on the right-hand side of the conditional, which we already assume is restricted to mentioning only unary predicates, its complexity is independent of the bound ρ.

The following theorem, due to Lewis [30], is the key to proving the lower bound for well-definedness (and for some of the other results in this section as well).

THEOREM 4.10. [30] *The problem of deciding whether a sentence $\xi \in \mathcal{L}^{-}(\mathcal{Q})$ is satisfiable is NEXPTIME-complete. Moreover, the lower bound holds even for formulas ξ of depth* 2.

Lewis proves this as follows: for any nondeterministic Turing machine **M** that runs in exponential time and any input w, he constructs a sentence $\xi \in \mathcal{L}^{-}(\mathcal{Q})$ of quantifier depth 2 and whose length is polynomial in the size of **M** and w, such that ξ is satisfiable iff there is an accepting computation of **M** on w.

Our first use of Lewis's result is to show that determining well-definedness is NEXPTIME-complete; this result does not require the assumptions that we are making throughout the rest of this section.

THEOREM 4.11. *For $\theta \in \mathcal{L}(\Upsilon)$, the problem of deciding if* $\Pr^{w}_{\infty}(* \mid \theta)$ *is well defined is NEXPTIME-complete. The NEXPTIME lower bound holds even for $\theta \in \mathcal{L}^{-}(\mathcal{Q})$ where $d(\theta) \leq 2$.*

Proof. For the upper bound, we proceed much as in Theorem 4.6. Let $\Psi = \Upsilon_\theta$ and let $\mathcal{C} = \mathcal{D}_\theta$. We know that $\Pr^{w}_{\infty}(* \mid \theta)$ is well defined iff there exists an atomic description ψ of size $M = d(\theta) + |\mathcal{C}|$ over Ψ and some model fragment $\mathcal{V} \in \mathcal{M}^{\Psi}(\psi)$ such that $\alpha(\psi) > 0$ and $\Pr^{w}_{\infty}(\theta \mid \psi \wedge \mathcal{V}) = 1$. Since all the predicates in Ψ have arity 1, it follows from Proposition 4.5 that the size of a model description $\psi \wedge \mathcal{V}$ over Ψ is $O(|\Psi| 2^{|\mathcal{P}|} M)$. Since $|\Psi| < |\theta|$, this implies that model descriptions have exponential length, and can be generated by a nondeterministic exponential time Turing machine. Because we can assume that $\rho = 1$ here when applying Corollary 4.4, we can also deduce that we can check whether $\Pr^{w}_{\infty}(\theta \mid \psi \wedge \mathcal{V})$ is 0 or 1 using a deterministic Turing machine in time $2^{O(|\theta||\Psi| \log(\nu(\psi)+1))}$. Since $|\Psi| \leq |\theta|$, and $\nu(\psi)$ is at most exponential in $|\theta|$, it follows that we can decide if $\Pr^{w}_{\infty}(\theta \mid \psi \wedge \mathcal{V}) = 1$ in deterministic time exponential in $|\theta|$. Thus, to check if $\Pr^{w}_{\infty}(* \mid \theta)$ is well defined we nondeterministically guess a model description $\psi \wedge \mathcal{V}$ of the right type, and check that $\alpha(\psi) > 0$ and that $\Pr^{w}_{\infty}(\theta \mid \psi \wedge \mathcal{V}) = 1$. The entire procedure can be executed in nondeterministic exponential time.

For the lower bound, observe that if a formula ξ in $\mathcal{L}^-(\Phi)$ is satisfied in some model with domain $\{1, \dots, N\}$ then it is satisfiable in some model of every domain size larger than N. Therefore, $\xi \in \mathcal{L}^-(\mathcal{Q})$ is satisfiable if and only if the limit $\Pr^w_\infty(* \mid \xi)$ is well defined. The result now follows from Theorem 4.10. □

We next consider the problem of computing the asymptotic probability $\Pr^w_\infty(\varphi \mid \theta)$, given that it is well defined. We show that this problem is #EXP-complete. Recall that #P (see [38]) is the class of integer functions computable as the number of accepting computations of a nondeterministic polynomial-time Turing machine. More precisely, a function $f : \{0, 1\}^* \rightarrow \mathbb{N}$ is said to be in #P if there is a nondeterministic Turing machine $\mathbf{M}$ such that for any w, the number of accepting paths of $\mathbf{M}$ on input w is $f(w)$. The class #EXP is the exponential time analogue.

The function we are interested in is $\Pr^w_\infty(\varphi \mid \theta)$, which is not integer valued. Nevertheless, we want to show that it is in #EXP. In the spirit of similar definitions for #P (see, for example, [39] and [34]) and NP (e.g., [17]) we extend the definition of #EXP to apply also to non-integer-valued functions.

DEFINITION 4.12. An arbitrary function f is said to be *#EXP-easy* if there exists an integer-valued function g in #EXP and a polynomial-time-computable function h such that for all x, $f(x) = h(g(x))$. (In particular, we allow h to involve divisions, so that $f(x)$ may be a rational function.) A function f is *#EXP-hard* if, for every #EXP-easy function g, there exist polynomial-time functions h_1 and h_2 such that, for all x, $g(x) = h_2(f(h_1(x)))$.[10] A function f is *#EXP-complete* if it is #EXP-easy and #EXP-hard. □

We can similarly define analogues of these definitions for the class #P.

We now show that for an infinite arity-bounded vocabulary in which equality is not used, or for any unary vocabulary, the problem of computing the asymptotic conditional probability is #EXP-complete. We start with the upper bound.

THEOREM 4.13. *If either* (a) $\varphi, \theta \in \mathcal{L}(\Upsilon)$ *or* (b) $\varphi \in \mathcal{L}(\Omega)$ *and* $\theta \in \mathcal{L}^-(\Upsilon)$, *then computing* $\Pr^w_\infty(\varphi \mid \theta)$ *is #EXP-easy.*

Proof. Let $\Phi = \Omega_{\varphi\wedge\theta}$, let $\Psi = \Upsilon_{\varphi\wedge\theta}$, and let $\mathcal{P}$ and $\mathcal{C}$ be the appropriate subsets of Ψ. Let $\delta_\theta = \Delta^\Psi(\theta)$. Recall from the proof of Theorem 4.7 that we would like to generate the model descriptions $\psi \wedge \mathcal{V}$ of degree δ_θ, consider the ones for which $\Pr^w_\infty(\theta \mid \psi \wedge \mathcal{V}) = 1$, and compute the fraction of those for which $\Pr^w_\infty(\varphi \mid \psi \wedge \mathcal{V})$. More precisely, consider the set of model descriptions of size $M = d(\varphi \wedge \theta) + |\mathcal{C}|$. For a degree δ, let $count^\delta(\theta)$ denote the number of those model descriptions for which $\Pr^w_\infty(\theta \mid \psi \wedge \mathcal{V}) = 1$. Similarly, let $count^\delta(\varphi)$ denote the number for which $\Pr^w_\infty(\varphi \wedge \theta \mid \psi \wedge \mathcal{V}) = 1$. We are interested in the value of the fraction $count^{\delta_\theta}(\varphi)/count^{\delta_\theta}(\theta)$.

We want to show that we can nondeterministically generate model descriptions $\psi \wedge \mathcal{V}$, and check in deterministic exponential time whether $\Pr^w_\infty(\theta \mid \psi \wedge \mathcal{V})$ (or, similarly, $\Pr^w_\infty(\varphi \wedge \theta \mid \psi \wedge \mathcal{V})$) is 0 or 1. We begin by showing the second part: that the 0-1 probabilities can be computed in deterministic exponential time. There are two cases to consider. In case (a), φ and θ are both unary, allowing us to assume that $\rho = 1$ for the purposes of Corollary 4.4. In this case, the 0-1 computations can be done in time $2^{O(|\varphi\wedge\theta||\Psi|\log(\nu(\psi)+1))}$, where $\Psi = \Upsilon_{\varphi\wedge\theta}$. As in Theorem 4.11, $|\Psi| \leq |\varphi \wedge \theta|$ and $\nu(\psi)$ is at most exponential in $|\theta|$, allowing us to carry out the computation in deterministic exponential time. In case (b), $\theta \in \mathcal{L}^-(\Upsilon)$, and therefore the only named elements are the constants. In this case, the 0-1 probabilities can be computed in deterministic time $2^{O(|\Phi|(\nu(\psi)+|\varphi\wedge\theta|)^\rho)}$, where $\Phi = \Omega_{\varphi\wedge\theta}$. However, as we have just discussed, $\nu(\psi) < |\varphi \wedge \theta|$, implying that the computation can be completed in exponential time.

[10] Notice that we need the function h_2 as well as h_1. For example, if g is an integer-valued function and f always returns a rational value between 0 and 1, as is the case for us, then there is no function h_1 such that $g(x) = f(h_1(x))$.

Having shown how the 0-1 probabilities can be computed, it remains only to generate model descriptions in the appropriate way. However, we do not want to consider all model descriptions, because we must count only those model descriptions of degree δ_θ. The problem is that we do not know δ_θ in advance. We will therefore construct a nondeterministic exponential time Turing machine $\mathbf{M}$ such that the number of accepting paths of $\mathbf{M}$ encodes, for each degree δ, both $\mathit{count}^{\delta}(\varphi)$ and $\mathit{count}^{\delta}(\theta)$. We need to do the encoding in such a way as to be able to isolate the counts for δ_θ when we finally know its value. This is done as follows.

Let ψ be an atomic description ψ over Ψ of size M. Recall that the degree $\Delta(\psi)$ is a pair $(\alpha(\psi), \nu(\psi))$ such that $\alpha(\psi) \leq 2^{|\mathcal{P}|}$ and $\nu(\psi) \leq 2^{|\mathcal{P}|}M$. Thus, there are at most $E = 2^{2|\mathcal{P}|}M$ possible degrees. Number the degrees in increasing order: $\delta_1, \ldots, \delta_E$. We want it to be the case that the number of accepting paths of $\mathbf{M}$ written in binary has the form

$$p_{10} \cdots p_{1m} q_{10} \cdots q_{1m} \cdots p_{E0} \cdots p_{Em} q_{E0} \cdots q_{Em},$$

where $p_{i0} \ldots p_{im}$ is the binary representation of $\mathit{count}^{\delta_i}(\varphi)$ and $q_{i0} \ldots q_{im}$ is the binary representation of $\mathit{count}^{\delta_i}(\theta)$. We choose m to be sufficiently large so that there is no overlap between the different sections of the output. The largest possible value of an expression of the form $\mathit{count}^{\delta_i}(\theta)$ is the maximum number of model descriptions of degree δ_i over Φ. This is clearly less than the overall number of model descriptions, which we computed in §4.2.

The machine $\mathbf{M}$ proceeds as follows. Let m be the smallest integer such that 2^m is more than the number of possible model descriptions, which, by Proposition 4.5, is $2^{O(|\Phi|(2^{|\mathcal{P}|}M)^{\rho})}$. Note that m is exponential and that $\mathbf{M}$ can easily compute m from Φ. $\mathbf{M}$ then nondeterministically chooses a degree δ_i, for $i = 1, \ldots, E$. It then executes $E - i$ phases, in each of which it nondeterministically branches $2m$ times. This has the effect of giving this branch a weight of $2^{2m(E-i)}$. It then nondeterministically chooses whether to compute $p_{i0} \ldots p_{im}$ or $q_{i0} \ldots q_{im}$. If the former, it again branches m times, separating the results for $\mathit{count}^{\delta_i}(\varphi)$ from those for $\mathit{count}^{\delta_i}(\theta)$. In either case, it now nondeterministically generates all model descriptions $\psi \wedge \mathcal{V}$ over Φ. It ignores those for which $\Delta(\psi) \neq \delta_i$. For the remaining model descriptions $\psi \wedge \mathcal{V}$, it computes $\Pr^w_\infty(\varphi \wedge \theta \mid \psi \wedge \mathcal{V})$ in the first case, and $\Pr^w_\infty(\theta \mid \psi \wedge \mathcal{V})$ in the latter. This is done in exponential time, using the same technique as in Theorem 4.11. The machine accepts precisely when this probability is 1.

This procedure is executable in nondeterministic exponential time, and results in the appropriate number of accepting paths. It is now easy to compute $\Pr^w_\infty(\varphi \mid \theta)$ by finding the largest degree δ for which $\mathit{count}^{\delta}(\theta) > 0$, and dividing $\mathit{count}^{\delta}(\varphi)$ by $\mathit{count}^{\delta}(\theta)$. □

We now want to prove the matching lower bound. As in Theorem 4.11, we make use of Lewis's NEXPTIME-completeness result. As there, this allows us to prove the result even for $\varphi, \theta \in \mathcal{L}^-(\mathcal{Q})$ of quantifier depth 2. A straightforward modification of Lewis's proof shows that, given w and a nondeterministic exponential time Turing machine $\mathbf{M}$, we can construct a depth-2 formula $\xi \in \mathcal{L}^-(\mathcal{Q})$ such that the number of simplified atomic descriptions over $\mathcal{P}_\xi$ consistent with ξ is exactly the number of accepting computations of $\mathbf{M}$ on w. This allows us to prove the following theorem.

THEOREM 4.14. *Given $\xi \in \mathcal{L}^-(\mathcal{Q})$, counting the number of simplified atomic descriptions over $\mathcal{P}_\xi$ consistent with ξ is #EXP-complete. The lower bound holds even for formulas ξ of depth* 2.

This theorem forms the basis for our own hardness result.

THEOREM 4.15. *Given $\varphi, \theta \in \mathcal{L}^-(\mathcal{Q})$ of depth at least* 2, *the problem of computing* $\Pr^w_\infty(\varphi \mid \theta)$ *is #EXP-hard, even given an oracle for deciding whether the limit exists.*

Proof. Given $\varphi \in \mathcal{L}^-(\mathcal{Q})$, we reduce the problem of counting the number of simplified atomic descriptions over $\mathcal{P}_\varphi$ consistent with φ to that of computing an appropriate asymptotic probability. Recall that, for the language $\mathcal{L}^-(\mathcal{Q})$, model descriptions are equivalent to sim-

A_1	A_2	A_3	A_4
$*$	0	$*$	0
$*$	$*$	0	$*$

FIG. 2. *Two atomic descriptions with different degrees.*

	A_1	A_2	A_3	A_4
$\wedge Q:$	$*$	0	$*$	0
$\wedge\neg Q:$	0	$*$	0	$*$
$\wedge Q:$	$*$	$*$	0	$*$
$\wedge\neg Q:$	0	0	$*$	0

FIG. 3. *Two maximal atomic descriptions.*

plified atomic descriptions. Therefore, computing an asymptotic conditional probability for this language reduces to counting simplified atomic descriptions of maximal degree. Thus, the major difficulty we need to overcome here is the converse of the difficulty that arose in the upper bound. We now want to count *all* simplified atomic descriptions consistent with φ, while using the asymptotic conditional probability in the most obvious way would only let us count those of maximum degree. For example, the two atomic descriptions whose characteristics are represented in Fig. 2 have different degrees; the first one will thus be ignored by a computation of asymptotic conditional probabilities.

Let $\mathcal{P}$ be $\mathcal{P}_\varphi = \{P_1, \ldots, P_k\}$, and let Q be a new unary predicate not in $\mathcal{P}$. We let $A_1, \ldots, A_K$ for $K = 2^k$ be all the atoms over $\mathcal{P}$, and let $A'_1, \ldots, A'_{2K}$ be all the atoms over $\mathcal{P}' = \mathcal{P} \cup \{Q\}$, such that $A'_i = A_i \wedge Q$ and $A'_{K+i} = A_i \wedge \neg Q$ for $i = 1, \ldots, K$.

We define θ' as follows:

$$\theta' =_{\text{def}} \forall x, y \left(\left(Q(x) \wedge \bigwedge_{i=1}^{k} (P_i(x) \Leftrightarrow P_i(y)) \right) \Rightarrow Q(y) \right).$$

The sentence θ' guarantees that the predicate Q is "constant" on the atoms defined by $\mathcal{P}$. That is, if A_i is an atom over $\mathcal{P}$, it is not possible to have $\exists x\,(A_i(x) \wedge Q(x))$ as well as $\exists x\,(A_i(x) \wedge \neg Q(x))$. Therefore, if ψ is a simplified atomic description over $\mathcal{P}'$ which is consistent with θ', then, for each $i \leq K$, at most one of the atoms A'_i and A'_{K+i} can be active, while the other is necessarily empty. It follows that $\alpha(\psi) \leq K$. Since there are clearly atomic descriptions of activity count K consistent with θ', the atomic descriptions of maximal degree are precisely those for which $\alpha(\psi) = K$. Moreover, if $\alpha(\psi) = K$, then A'_i is active iff A'_{K+i} is not. Two atomic descriptions of maximal degree are represented in Fig. 3. Thus, for each set $I \subseteq \{1, \ldots, K\}$, there is precisely one simplified atomic description ψ consistent with θ' of activity count K where A'_i is active in ψ iff $i \in I$. Therefore, there are exactly 2^K simplified atomic descriptions ψ over $\mathcal{P}'$ consistent with θ' for which $\alpha(\psi) = K$.

Let $\theta = \theta' \wedge \exists x\, Q(x)$. Notice that all simplified atomic descriptions ψ with $\alpha(\psi) = K$ that are consistent with θ' are also consistent with θ, except for the one where no atom in $A'_1, \ldots, A'_K$ is active. Thus, $|\mathcal{A}_\theta^{\mathcal{P}',K}| = 2^K - 1$. For the purposes of this proof, we call a simplified atomic description ψ over $\mathcal{P}'$ consistent with θ for which $\alpha(\psi) = K$ a *maximal atomic description.* Notice that there is an obvious one-to-one correspondence between consistent simplified atomic descriptions over $\mathcal{P}$ and maximal atomic descriptions over $\mathcal{P}'$. A maximal atomic description where A'_i is active iff $i \in I$ (and A'_{K+i} is active for $i \notin I$) corresponds to the simplified atomic description over $\mathcal{P}$ where A_i is active iff $i \in I$. (For example, the two consistent simplified atomic descriptions over $\{P_1, P_2\}$ in Fig. 2 correspond to the two maximal atomic descriptions over $\{P_1, P_2, Q\}$ in Fig. 3.) In fact, the reason we consider θ rather than θ' is precisely because there is no consistent simplified atomic description over $\mathcal{P}$ which corresponds to the maximal atomic description where no atom in $A'_1, \ldots, A'_K$ is active (since there is no consistent atomic description over $\mathcal{P}$ where none of $A_1, \ldots, A_K$ are active). Thus, we have overcome the hurdle discussed above.

We now define φ_Q; intuitively, φ_Q is φ restricted to elements that satisfy Q. Formally, we define ξ_Q for any formula ξ by induction on the structure of the formula:

- $\xi_Q = \xi$ for any atomic formula ξ,
- $(\neg\xi)_Q = \neg\xi_Q$,
- $(\xi \wedge \xi')_Q = \xi_Q \wedge \xi'_Q$,
- $(\forall y\, \xi(y))_Q = \forall y\, (Q(y) \Rightarrow \xi_Q(y))$.

Note that the size of φ_Q is linear in the size of φ. The one-to-one mapping discussed above from simplified atomic descriptions to maximal atomic descriptions gives us a one-to-one mapping from simplified atomic descriptions over $\mathcal{P}$ consistent with φ to maximal atomic descriptions consistent with $\varphi_Q \wedge \exists x\, Q(x)$. This is true because a model satisfies φ_Q iff the same model restricted to elements satisfying Q satisfies φ. Thus, the number of model descriptions over $\mathcal{P}$ consistent with φ is precisely $|\mathcal{A}^{\mathcal{P}',K}_{\varphi_Q \wedge \theta}|$.

From Corollary 3.36, it follows that

$$\Pr{}^w_\infty(\varphi_Q \mid \theta) = \frac{|\mathcal{A}^{\mathcal{P}',K}_{\varphi_Q \wedge \theta}|}{|\mathcal{A}^{\mathcal{P}',K}_{\theta}|} = \frac{|\mathcal{A}^{\mathcal{P}}_{\varphi}|}{2^K - 1}.$$

Thus, the number of simplified atomic descriptions over $\mathcal{P}$ consistent with φ is $(2^K - 1)\Pr^w_\infty(\varphi_Q \mid \theta)$. This proves the lower bound. □

As in Theorem 4.8, we can also show that any nontrivial approximation of the asymptotic probability is hard, even if we restrict to sentences of depth 2.

THEOREM 4.16. *Fix rational numbers* $0 \leq r_1 \leq r_2 \leq 1$ *such that* $[r_1, r_2] \neq [0, 1]$. *For* $\varphi, \theta \in \mathcal{L}^-(\mathcal{Q})$ *of depth at least* 2, *the problem of deciding whether* $\Pr^w_\infty(\varphi \mid \theta) \in [r_1, r_2]$ *is both NEXPTIME-hard and co-NEXPTIME-hard, even given an oracle for deciding whether the limit exists.*

Proof. Let us begin with the case where $r_1 = 0$ and $r_2 < 1$. Consider any $\varphi \in \mathcal{L}^-(\mathcal{Q})$ of depth at least 2, and assume without loss of generality that $\mathcal{P} = \mathcal{P}_\varphi = \{P_1, \ldots, P_k\}$. Choose $Q \notin \mathcal{P}$, let $\mathcal{P}' = \mathcal{P} \cup \{Q\}$, and let ξ be $\forall x(P_1(x) \wedge \ldots \wedge P_k(x) \wedge Q(x))$. We consider $\Pr^w_\infty(\varphi \mid \varphi \vee \xi)$. Clearly $\varphi \vee \xi$ is satisfiable, so that this asymptotic probability is well defined. If φ is unsatisfiable, then $\Pr^w_\infty(\varphi \mid \varphi \vee \xi) = 0$. On the other hand, if φ is satisfiable, then $\alpha^{\mathcal{P}}(\varphi) = j > 0$ for some j. It is easy to see that $\alpha^{\mathcal{P}'}(\varphi) = \alpha^{\mathcal{P}'}(\varphi \vee \xi) = 2j$. Moreover, φ and $\varphi \vee \xi$ are consistent with precisely the same simplified atomic descriptions with $2j$ active atoms. This is true since $\alpha^{\mathcal{P}'}(\xi) = 1 < 2j$. It follows that if φ is satisfiable, then $\Pr^w_\infty(\varphi \mid \varphi \vee \xi) = 1$.

Thus, we have that $\Pr^w_\infty(\varphi \mid \varphi \vee \xi)$ is either 0 or 1, depending on whether or not φ is satisfiable. Thus, $\Pr^w_\infty(\neg\varphi \mid \varphi \vee \xi)$ is in $[r_1, r_2]$ iff φ is satisfiable; similarly, $\Pr^w_\infty(\neg\varphi \mid \neg\varphi \vee \xi)$ is in $[r_1, r_2]$ iff φ is valid. By Theorem 4.10, it follows that this approximation problem is both NEXPTIME-hard and co-NEXPTIME-hard.

If $r_1 = q/p > 0$, we construct sentences φ_{r_1} and θ_{r_1} of depth 2 in $\mathcal{L}^-(\mathcal{Q})$ such that $\Pr^w_\infty(\varphi_{r_1} \mid \theta_{r_1}) = r_1$.[11] Choose $\ell = \lceil \log p \rceil$, and let $\mathcal{P}'' = \{Q_1, \ldots, Q_\ell\}$ be a set of predicates

[11] The sentences constructed in Theorem 4.8 for the same purpose will not serve our purpose in this theorem, since they used unbounded quantifier depth.

such that $\mathcal{P}'' \cap \mathcal{P}' = \emptyset$. Let $A_1, \ldots, A_{2^\ell}$ be the set of atoms over $\mathcal{P}''$. We define θ_{r_1} to be

$$\exists^1 x\,(A_1(x) \vee A_2(x) \vee \ldots \vee A_p(x)).$$

Similarly, φ_{r_1} is defined as

$$\exists^1 x\,(A_1(x) \vee A_2(x) \vee \ldots \vee A_q(x)).$$

Recall from §3.1 that the construct "$\exists^1 x$" can be defined in terms of a formula of quantifier depth 2. There are exactly p atomic descriptions of size 2 of maximal degree consistent with θ_{r_1}; each has one element in one of the atoms $A_1, \ldots, A_p$ and no elements in the rest of the atoms among $A_1, \ldots, A_p$, with all the remaining atoms (those among $A_{p+1}, \ldots, A_{2^\ell}$) being active. Among these atomic descriptions, q are also consistent with φ_{r_1}. Therefore, $\Pr^w_\infty(\varphi_{r_1} \mid \theta_{r_1}) = q/p$. Since the predicates occurring in $\varphi_{r_1}, \theta_{r_1}$ are disjoint from $\mathcal{P}'$, it follows that

$$\Pr^w_\infty(\varphi \wedge \varphi_{r_1} \mid (\varphi \vee \xi) \wedge \theta_{r_1}) = \Pr^w_\infty(\varphi \mid \varphi \vee \xi) \cdot \Pr^w_\infty(\varphi_{r_1} \mid \theta_{r_1}) = \Pr^w_\infty(\varphi \mid \varphi \vee \xi) \cdot r_1.$$

This is equal to r_1 (and hence is in $[r_1, r_2]$) if and only if φ is satisfiable, and is 0 otherwise. □

The lower bounds in this section all hold provided we consider formulas whose quantification depth is at least 2. Can we do better if we restrict to formulas of quantification depth at most 1? As is suggested by Table 1, we can. The complexities typically drop by an exponential factor. For example, checking well-definedness becomes NP-complete rather than NEXPTIME-complete. For the problem of computing probabilities for formulas with quantification depth 1, we know that the problem is in PSPACE, and is (at least) #P-hard. Finally, the problem of approximating probabilities is hard for both NP and co-NP. A detailed analysis of these results can be found in [27]; some related work for a propositional language has been done by Roth [35].

4.5. Infinite vocabulary—the general case. In §4.4 we investigated the complexity of asymptotic conditional probabilities when the (infinite) vocabulary satisfies certain restrictions. As we now show, the results there were tight in the sense that the restrictions cannot be weakened. We examine the complexity of the general case, in which the vocabulary is infinite with no bound on predicates' arities and/or in which equality can be used.

The problem of checking if $\Pr^w_\infty(\varphi \mid \theta)$ is well defined is still NEXPTIME-complete. Theorem 4.11 (which had no restrictions) still applies. However, the complexity of the other problems we consider does increase. It can be best described in terms of the complexity class TA(EXP,LIN)—the class of problems that can be solved by an exponential time ATM with a linear number of alternations. The class TA(EXP,LIN) also arises in the study of unconditional probabilities where there is no bound on the arity of the predicates. Grandjean [20] proved a TA(EXP,LIN) upper bound for computing whether the unconditional probability is 0 or 1 in this case, and Immerman [4] proved a matching lower bound. Of course, Grandjean's result can be viewed as a corollary of Theorem 4.3. Immerman's result, which has not, to the best of our knowledge, appeared in print, is a corollary of Theorem 4.18 which we prove in this section.

To capture the complexity of computing the asymptotic probability in the general case, we use a counting class #TA(EXP,LIN) that corresponds to TA(EXP,LIN). To define this class, we restrict attention to the class of ATMs whose initial states are existential. Given such an ATM **M**, we define an *initial existential path* in the computation tree of **M** on input w to be a path in this tree, starting at the initial state, such that every node on the path corresponds to an existential state except for the last node, which corresponds to a universal or an accepting state. That is, an initial existential path is a maximal path that starts at the root of the tree and contains only existential nodes except for the last node in the path. We say that an integer-valued function $f : \{0, 1\}^* \to I\!N$ is in #TA(EXP,LIN) if there is a machine **M** in the class TA(EXP,LIN) such that, for all w, $f(w)$ is the number of existential paths in the computation tree of **M** on input w whose last node is accepting (recall that we defined a notion of "accepting" for any node in the tree in §4.1). We extend the definition of #TA(EXP,LIN) to apply to non-integer-valued functions and define *#TA(EXP,LIN)-easy* just as we did before with #P and #EXP in §4.4.

We start with the upper bound.

THEOREM 4.17. *For* $\varphi \in \mathcal{L}(\Omega)$ *and* $\theta \in \mathcal{L}(\Upsilon)$*, the function* $\Pr^w_\infty(\varphi \mid \theta)$ *is in #TA(EXP,LIN).*

Proof. Let $\Phi = \Omega_{\varphi\wedge\theta}$, let $\Psi = \Upsilon_{\varphi\wedge\theta}$, and let ρ be the maximum arity of a predicate in Φ. The proof proceeds precisely as in Theorem 4.13. We compute, for each degree δ, the values $count^\delta(\theta)$ and $count^\delta(\varphi)$. This is done by nondeterministically generating model descriptions $\psi \wedge \mathcal{V}$ over Φ, branching according to the degree of ψ, and computing $\Pr^w_\infty(\varphi \wedge \theta \mid \psi \wedge \mathcal{V})$ and $\Pr^w_\infty(\theta \mid \psi \wedge \mathcal{V})$ using a TA(EXP,LIN) Turing machine.

To see that this is possible, recall from Proposition 4.5 that the length of a model description over Φ is $O(|\Phi|(2^{|\mathcal{P}|}M)^\rho)$. This is exponential in $|\Phi|$ and ρ, both of which are at most $|\varphi \wedge \theta|$. Therefore, it is possible to guess a model description in exponential time. Similarly, as we saw in the proof of Theorem 4.13, only exponentially many nondeterministic guesses are required to separate the output so that counts corresponding to different degrees do not overlap. These guesses form the initial nondeterministic stage of our TA(EXP,LIN) Turing machine. Note that it is necessary to construct the rest of the Turing machine so that a universal state always follows this initial stage, so that each guess corresponds exactly to one initial existential path; however, this is easy to arrange.

For each model description $\psi \wedge \mathcal{V}$ so generated, we compute $\Pr^w_\infty(\theta \mid \psi \wedge \mathcal{V})$ or $\Pr^w_\infty(\varphi \wedge \theta \mid \psi \wedge \mathcal{V})$ as appropriate, accepting if the conditional probability is 1. It follows immediately from Theorem 4.3 and the fact that there can only be exponentially many named elements in any model description we generate that this computation is in TA(EXP,LIN). Thus, the problem of computing $\Pr^w_\infty(\varphi \mid \theta)$ is in #TA(EXP,LIN). □

We now want to prove the matching lower bound. Moreover, we would like to show that the restrictions from §4.4 cannot be weakened. Recall from Theorem 4.13 that the #EXP upper bound held under one of two conditions: either (a) φ and θ are both unary, or (b) the vocabulary is arity-bounded and θ does not use equality. To show that (a) is tight, we show that the #TA(EXP,LIN) lower bound holds even if we allow φ and θ to use only binary predicates and equality. (The use of equality is necessary, since without it we know from (b) that the problem is #EXP-easy.) To show that (b) is tight, we show that the lower bound holds for a non-arity-bounded vocabulary, but without allowing equality in θ. Neither lower bound requires the use of constants.

The proof of the lower bounds is lengthy, but can be simplified somewhat by some assumptions about the construction of the TA(EXP,LIN) machines we consider. The main idea is that the existential "guesses" being made in the initial phase should be clearly distinguished from the rest of the computation. To achieve this, we assume that the Turing machine has an additional *guess tape*, and the initial phase of every computation consists of nondeterministically generating a *guess string* γ which is written on the new tape. The machine then proceeds with a standard alternating computation, but with the possibility of reading the bits on the guess tape.

More precisely, from now on we make the following assumptions about an ATM **M**. Consider any increasing functions $T(n)$ and $A(n)$ (in essence, these correspond to the time complexity and number of alternations), and let w be an input of size n. We assume:

- **M** has two tapes and two heads (one for each tape). Both tapes are one-way infinite to the right.
- The first tape is a work tape, which initially contains only the string w.
- **M** has an initial nondeterministic phase, during which its only action is to nondeterministically generate a string γ of zeros and ones, and write this string on the second tape (the guess tape). The string γ is always of length $T(n)$. Moreover, at the end of this phase, the work tape is as in the initial configuration, the guess tape contains only γ, the heads are at the beginning of their respective tapes, and the machine is in a distinguished universal state s_0.
- After the initial phase, the guess tape is never changed.
- After the initial phase, **M** takes at most $T(n)$ steps on each branch of its computation tree, and makes exactly $A(n) - 1$ alternations before entering a terminal (accepting or rejecting) state.
- The state before entering a terminal state is always an existential state (i.e., $A(n)$ is odd).

Let **M**$'$ be any (unrestricted) TA(T,A) machine that "computes" an integer function f. It is easy to construct some **M** satisfying the restrictions above that also computes f. The machine **M** first generates the guess string γ, and then simulates **M**$'$. At each nondeterministic branching point in the initial existential phase of **M**$'$, **M** uses the next bit of the string γ to dictate which choice to take. Observe that this phase is deterministic (given γ), and can thus be folded into the following universal phase. (Deterministic steps can be viewed as universal steps with a single successor.) If not all the bits in γ are used, **M** continues the execution of **M**$'$, but checks in parallel that the unused bits of γ are all 0's. If not, **M** rejects. It is easy to see that on any input w, **M** has the same number of accepting paths as **M**$'$, and therefore accepts the same function f. Moreover, **M** has the same number of alternations as **M**$'$, and at most a constant factor blowup in the running time.[12] This shows that it will be sufficient to prove our hardness results for the class #TA(EXP,LIN) by considering only those machines that satisfy these restrictions. For the remainder of this section we will therefore assume that all ATMs are of this type.

Let **M** be such an ATM and let w be an input of size n. We would like to encode the computation of **M** on w using a pair of formulas φ_w, θ_w. (Of course, these formulas depend on **M** as well, but we suppress this dependence.) Our first theorem shows how to encode part of this computation: given some appropriate string γ of length $T(n)$, we construct formulas that encode the computation of **M** immediately following the initial phase of guessing γ. More precisely, we say that **M** *accepts* w *given* γ if, on input w, the initial existential path during which **M** writes γ on the guess tape leads to an accepting node. We construct formulas $\varphi_{w,\gamma}$ and $\theta_{w,\gamma}$ such that $\Pr_\infty^w(\varphi_{w,\gamma} \mid \theta_{w,\gamma})$ is either 0 or 1, and is equal to 1 iff **M** accepts w given γ.

We do not immediately want to specify the process of guessing γ, so our initial construction will not commit to this. For a predicate R, let $\varphi[R]$ be a formula that uses the predicate R. Let ξ be another formula that has the same number of free variables as the arity of R. Then $\varphi[\xi]$ is the formula where every occurrence of R is replaced with the formula ξ, with an appropriate substitution of the arguments of R for the free variables in ξ.

THEOREM 4.18. *Let* **M** *be a TA(T,A) machine as above, where $T(n) = 2^{t(n)}$ for some polynomial $t(n)$ and $A(n) = O(n)$. Let w be an input string of length n, and $\gamma \in \{0,1\}^{T(n)}$ be a guess string.*

[12] For ease of presentation, we can and will (somewhat inaccurately, but harmlessly) ignore this constant factor and say that the time complexity of **M** is, in fact, $T(n)$.

(a) *For a unary predicate R, there exist formulas $\varphi_w[R]$, $\xi_\gamma \in \mathcal{L}(\Omega)$ and $\theta_w \in \mathcal{L}(\Upsilon)$ such that* $\Pr^w_\infty(\varphi_w[\xi_\gamma] \mid \theta_w) = 1$ *iff* $\mathbf{M}$ *accepts w given γ and is* 0 *otherwise. Moreover, φ_w uses only predicates with arity* 2 *or less.*

(b) *For a binary predicate R, there exist formulas $\varphi'_w[R]$, $\xi'_\gamma \in \mathcal{L}(\Omega)$ such that* $\Pr^w_\infty(\varphi'_w[\xi'_\gamma] \mid$ *true*$) = 1$ *iff* $\mathbf{M}$ *accepts w given γ and is* 0 *otherwise.*

The formulas $\varphi_w[R]$, θ_w, and $\varphi'_w[R]$ are independent of γ, and their length is polynomial in the representation of $\mathbf{M}$ *and w. Moreover, none of the formulas constructed use any constant symbols.*

Proof. Let Γ be the tape alphabet of $\mathbf{M}$ and let S be the set of states of $\mathbf{M}$. We will identify an instantaneous description (ID) of length ℓ of $\mathbf{M}$ with a string Σ^ℓ for $\Sigma = \Sigma_W \times \Sigma_G$, where Σ_W is $\Gamma \cup (\Gamma \times S)$ and Σ_G is $(\{0, 1\} \cup (\{0, 1\} \times \{h\}))$. We think of the Σ_W component of the ith element in a string as describing the contents of the ith location in the work tape and also, if the tape head is at location i, the state of the Turing machine. The Σ_G component describes the contents of the ith location in the guess tape (whose alphabet is $\{0, 1\}$) and whether the guess tape's head is positioned there. Of course, we consider only strings in which exactly one element in $\Gamma \times S$ appears in the first component and exactly one element in $\{0, 1\} \times \{h\}$ appears in the second component. Since $\mathbf{M}$ halts within $T(n)$ steps (not counting the guessing process, which we treat separately), we need only deal with IDs of length at most $T(n)$. Without loss of generality, assume all IDs have length exactly $T(n)$. (If necessary, we can pad shorter IDs with blanks.)

In both parts of the theorem, IDs are encoded using the properties of domain elements. In both cases, the vocabulary contains predicates whose truth value with respect to certain combinations of domain elements represent IDs. The only difference between parts (a) and (b) is in the precise encoding used. We begin by showing the encoding for part (a).

In part (a), we use the sentence θ_w to define $T(n)$ named elements. This is possible since θ_w is allowed to use equality. Each ID of the machine will be represented using a single domain element. The properties of the ID will be encoded using the relations between the domain element representing it and the named elements. More precisely, assume that the vocabulary has $t(n)$ unary predicates $P_1, \ldots, P_{t(n)}$, and one additional unary predicate P^*. The domain is divided into two parts: the elements satisfying P^* are the named elements used in the process of encoding IDs, while the elements satisfying $\neg P^*$ are used to actually represent IDs. The formula θ_w asserts (using equality) that each of the atoms A over $\{P^*, P_1, \ldots, P_{t(n)}\}$ in which P^* (as opposed to $\neg P^*$) is one of the conjuncts contains precisely one element:

$$\forall x, y \left(\left(P^*(x) \land P^*(y) \land \bigwedge_{i=1}^{t(n)} (P_i(x) \Leftrightarrow P_i(y)) \right) \Rightarrow x = y \right).$$

Note that θ_w has polynomial length and is independent of γ.

We can view an atom A over $\{P^*, P_1, \ldots, P_{t(n)}\}$ in which P^* is one of the conjuncts as encoding a number between 0 and $T(n) - 1$, written in binary: if A contains P_j rather than $\neg P_j$, then the jth bit of the encoded number is 1; otherwise it is 0. (Recall that $T(n)$, the running time of $\mathbf{M}$, is $2^{t(n)}$.) In the following, we let A_i, for $i = 0, \ldots, T(n) - 1$, denote the atom corresponding to the number i according to this scheme. Let e_i be the unique element in the atom A_i for $i = 0, \ldots, T(n) - 1$. When representing an ID using a domain element d (where $\neg P^*(d)$), the relation between d and e_i is used to represent the ith coordinate in the ID represented by d. Assume that the vocabulary has a binary predicate R_σ for each $\sigma \in \Sigma$. Roughly speaking, we say that the domain element d represents the ID $\sigma_0 \ldots \sigma_{T(n)-1}$ if $R_{\sigma_i}(d, e_i)$ holds for $i = 0, \ldots, T(n) - 1$. More precisely, we say that d

represents $\sigma_0 \ldots \sigma_{T(n)-1}$ if

$$\neg P^*(d) \wedge \bigwedge_{i=0}^{T(n)-1} \forall y \left(A_i(y) \Rightarrow \left(R_{\sigma_i}(d, y) \wedge \bigwedge_{\sigma' \in \Sigma - \{\sigma_i\}} \neg R_{\sigma'}(d, y) \right) \right).$$

Note that not every domain element d such that $\neg P^*(d)$ holds encodes a valid ID. However, the question of which ID, if any, is encoded by a domain element d depends only on the relations between d and the finite set of elements $e_0, \ldots, e_{T(n)-1}$. This implies that, with asymptotic probability 1, every ID will be encoded by some domain element. More precisely, let $ID(x) = \sigma_0 \ldots \sigma_{T(n)-1}$ be a formula which is true if x denotes an element that represents $\sigma_0 \ldots \sigma_{T(n)-1}$. (It should be clear that such a formula is indeed expressible in our language.) Then for each ID $\sigma_0 \ldots \sigma_{T(n)-1}$ we have

$$\Pr_\infty^w(\exists x\, (ID(x) = \sigma_0 \ldots \sigma_{T(n)-1}) \mid \theta_w) = 1.$$

For part (b) of the theorem, we must represent IDs in a different way because we are not allowed to condition on formulas that use equality. Therefore, we cannot create an exponential number of named elements using a polynomial-sized formula. The encoding we use in this case uses two domain elements per ID rather than one. We now assume that the vocabulary Ω contains a $t(n)$-ary predicate R_σ for each symbol $\sigma \in \Sigma$. Note that this uses the assumption that there is no bound on the arity of predicates in Ω. For $i = 0, \ldots, T(n) - 1$, let $b^i_{t(n)} \cdots b^i_1$ be the binary encoding of i. We say that the pair (d_0, d_1) of domain elements *represents the ID* $\sigma_0 \ldots \sigma_{T(n)-1}$ if

$$d_0 \neq d_1 \wedge \bigwedge_{i=0}^{T(n)-1} \left(R_{\sigma_i}(d_{b^i_1}, \ldots, d_{b^i_{t(n)}}) \wedge \bigwedge_{\sigma' \in \Sigma - \{\sigma_i\}} \neg R_{\sigma'}(d_{b^i_1}, \ldots, d_{b^i_{t(n)}}) \right).$$

Again, we can define a formula in our language $ID(x_0, x_1) = \sigma_0 \ldots \sigma_{T(n)-1}$ which is true if x_0, x_1 denote a pair of elements that represent $\sigma_0 \ldots \sigma_{T(n)-1}$. As before, observe that for each ID $\sigma_0 \ldots \sigma_{T(n)-1}$ we have

$$\Pr_\infty^w(\exists x_0, x_1\, (ID(x_0, x_1) = \sigma_0 \ldots \sigma_{T(n)-1}) \mid \mathit{true}) = 1.$$

In both case (a) and case (b), we can construct formulas polynomial in the size of $\mathbf{M}$ and w that assert certain properties. For example, in case (a), *Rep*(x) is true of a domain element d if and only if d encodes an ID. In this case, *Rep*(x) is the formula

$$\neg P^*(x) \wedge \forall y \left(P^*(y) \Rightarrow \dot{\bigvee}_{\sigma \in \Sigma} R_\sigma(x, y) \right) \wedge$$
$$\exists! y\, (P^*(y) \wedge \bigvee_{\sigma \in ((\Gamma \times S) \times \Sigma_G)} R_\sigma(x, y)) \wedge \exists! y\, (P^*(y) \wedge \bigvee_{\sigma \in (\Sigma_W \times (\{0,1\} \times \{h\}))} R_\sigma(x, y))$$

where $\dot{\bigvee}$ is an abbreviation whose meaning is that precisely one of its disjuncts is true.

In case (b), *Rep*(x_0, x_1) is true of a pair (d_0, d_1) if and only if it encodes an ID. The construction is similar. For instance, the conjunct of *Rep*(x_0, x_1) asserting that each tape position has a uniquely defined content is

$$x_0 \neq x_1 \wedge \forall z_1, \ldots, z_{t(n)} \left(\left(\bigwedge_{i=1}^{t(n)} (z_i = x_0 \vee z_i = x_1) \right) \Rightarrow_{\sigma \in \Sigma} a \dot{\bigvee_{\sigma \in \Sigma}} R_\sigma(z_1, \ldots, z_{t(n)}) \right).$$

Except for this assertion, the construction for the two cases is completely parallel given the encoding of IDs. We will therefore restrict the remainder of the discussion to case (a). Other relevant properties of an ID that we can formulate are:

- $Acc(x)$ (resp., $Univ(x)$, $Exis(x)$) is true of a domain element d if and only if d encodes an ID and the state in $ID(d)$ is an accepting state (resp., a universal state, an existential state).
- $Step(x, x')$ is true of elements d and d' if and only if both d and d' encode IDs and $ID(d')$ can follow from $ID(d)$ in one step of **M**.
- $Comp(x, x')$ is true of elements d and d' if and only if both d and d' encode IDs, and $ID(d')$ is the final ID in a maximal nonalternating path starting at $ID(d)$ in the computation tree of **M**, and the length of this path is at most $T(n)$. A maximal nonalternating path is either a path all of whose states are existential except for the last one (which must be universal or accepting), or a path all of whose states are universal except for the last one. We can construct *Comp* using a divide and conquer argument, so that its length is polynomial in $t(n)$.

We remark that *Acc*, *Step*, etc. are not new predicate symbols in the language. Rather, they are complex formulas described in terms of the basic predicates R_σ. We omit details of their construction here; these can be found in [20].

It remains only to describe the formula that encodes the initial configuration of **M** on input w. Since we are interested in the behavior of **M** given a particular guess string γ, we begin by encoding the computation of **M** after the initial nondeterministic phase; that is, after the string γ is already written on the guess tape and the rest of the machine is back in its original state. We now construct the formula $Init[R](x)$ that describes the initial configuration. This formula takes R as a parameter, and has the form $Init'(x) \wedge R(x)$. The formulas substituted for $R(x)$ will correspond (in a way discussed below) to possible guesses γ.

We begin by considering case (a). We assume the existence of an additional binary predicate B_0. It is easy to write a polynomial-length formula $Init'(x)$ which is true of a domain element d if and only if d represents an ID where: (a) the state is the distinguished state s_0 entered after the nondeterministic guessing phase, (b) the work tape contains only w, (c) the heads are at the beginning of their respective tapes, and (d) for all i, the ith location of the guess tape contains 0 iff $B_0(d, e_i)$. Here e_i is, as before, the unique element in atom A_i. Note that the last constraint can be represented polynomially using the formula

$$\forall y \left(P^*(y) \Rightarrow \left(B_0(x, y) \Leftrightarrow \bigvee_{\sigma \in \Sigma_W \times \{0,(0,h)\}} R_\sigma(x, y) \right) \right).$$

We also want to find a formula ξ_γ that can constrain B_0 to reflect the guess γ. This formula, which serves as a possible instantiation for R, does not have to be of polynomial size. We define it as follows, where for convenience, we use B_1 as an abbreviation for $\neg B_0$:

$$(1) \qquad \xi_\gamma(x) =_{\text{def}} \bigwedge_{i=0}^{T(n)-1} \forall y \left(A_i(y) \Rightarrow B_{\gamma_i}(x, y) \right) .$$

Note that this is of exponential length.

In case (b), the relation of the guess string γ to the initial configuration is essentially the same modulo the modifications necessary due to the different representation of IDs. We only sketch the construction. As in case (a), we add a predicate B'_0, but in this case of arity $t(n)$. Again, the predicate B'_0 represents the locations of the 0's in the guess tape following the initial nondeterministic phase. The specification of the denotation of this predicate is done using an exponential-sized formula ξ'_γ, as follows (again taking B'_1 to be an abbreviation for $\neg B'_0$):

$$\xi'_\gamma(x_0, x_1) =_{\text{def}} B'_{\gamma_0}(x_0, \ldots, x_0, x_0) \wedge B'_{\gamma_1}(x_0, \ldots, x_0, x_1) \wedge \ldots \wedge B'_{\gamma_{T(n)-1}}(x_1, \ldots, x_1, x_1).$$

Using these formulas, we can now write a formula expressing the assertion that **M** accepts w given γ. In writing these formulas, we make use of the assumptions made about **M** (that it is initially in the state immediately following the initial guessing phase, that all computation paths make exactly $A(n)$ alternations, and so on). The formula $\varphi_w[R]$ has the following form:

$$\exists x_1\,(\mathit{Init}[R](x_1) \wedge \forall x_2\,(\mathit{Comp}(x_1,x_2) \Rightarrow \exists x_3\,(\mathit{Comp}(x_2,x_3) \wedge \forall x_4\,(\mathit{Comp}(x_3,x_4) \Rightarrow \dots \\ \exists x_{A(n)}\,(\mathit{Comp}(x_{A(n)-1},x_{A(n)}) \wedge \mathit{Acc}(x_{A(n)}))\dots)))).$$

It is clear from the construction that $\varphi_w[R]$ does not depend on γ and that its length is polynomial in the representations of **M** and w.

Now suppose $\mathcal{W}$ is a world satisfying θ_w in which every possible ID is represented by at least one domain element. (As we remarked above, a random world has this property with asymptotic probability 1.) Then it is straightforward to verify that $\varphi_w[\xi_\gamma]$ is true in $\mathcal{W}$ iff **M** accepts w. Therefore $\Pr^w_\infty(\varphi_w[\xi_\gamma] \mid \theta_w) = 1$ iff **M** accepts w given γ and 0 otherwise. Similarly, in case (b), we have shown the construction of analogous formulas $\varphi'_w[R]$, for a binary predicate R, and ξ'_γ such that $\Pr^w_\infty(\varphi'_w[\xi'_\gamma] \mid \mathit{true}) = 1$ iff **M** accepts w given γ, and is 0 otherwise. □

We can now use the above theorem in order to prove the #TA(EXP,LIN) lower bound.

THEOREM 4.19. *For* $\varphi \in \mathcal{L}(\Omega)$ *and* $\theta \in \mathcal{L}(\Upsilon)$, *computing* $\Pr^w_\infty(\varphi \mid \theta)$ *is #TA(EXP,LIN)-hard. The lower bound holds even if* φ, θ *do not mention constant symbols and either* (a) φ *uses no predicate of arity* > 2, *or* (b) θ *uses no equality.*

Proof. Let **M** be a TA(EXP,LIN) Turing machine of the restricted type discussed earlier, and let w be an input of size n. We would like to construct formulas φ, θ such that from $\Pr^w_\infty(\varphi \mid \theta)$ we can derive the number of accepting computations of **M** on w. The number of accepting initial existential paths of such a Turing machine is precisely the number of guess strings γ such that **M** accepts w given γ. In Theorem 4.18, we showed how to encode the computation of such a machine **M** on input w given a nondeterministic guess γ. We now show how to force an asymptotic conditional probability to count guess strings appropriately.

As in Theorem 4.18, let $T(n) = 2^{t(n)}$, and let $\mathcal{P}' = \{P'_1, \dots, P'_{t(n)}\}$ be new unary predicates not used in the construction of Theorem 4.18. As before, we can view an atom A' over $\mathcal{P}'$ as representing a number in the range $0, \dots, T(n) - 1$: if A contains P'_j, then the jth bit of the encoded number is 1; otherwise it is 0. Again, let A'_i, for $i = 0, \dots, T(n) - 1$, denote the atom corresponding to the number i according to this scheme. We can view a simplified atomic description ψ over $\mathcal{P}'$ as representing the string $\gamma = \gamma_0 \dots \gamma_{T(n)-1}$ such that γ_i is 1 if ψ contains the conjunct $\exists z\, A'_i(z)$, and 0 if ψ contains its negation. Under this representation, for every string γ of length $T(n)$, there is a unique simplified atomic description over $\mathcal{P}'$ that represents it; we denote this atomic description ψ_γ. Note that ψ_γ is not necessarily a consistent atomic description, since the atomic description where all atoms are empty also denotes a legal string—that string where all bits are 0.

While it is possible to reduce the problem of counting accepting guess strings to that of counting simplified atomic descriptions, this is not enough. After all, we have already seen that computing asymptotic conditional probabilities ignores all atomic descriptions that are not of maximal degree. We deal with this problem as in Theorem 4.15. Let Q be a new unary predicate, and let θ' be, as in Theorem 4.15, the sentence

$$\forall x, y \left(\left(Q(x) \wedge \bigwedge_{j=1}^{t(n)} (P'_j(x) \Leftrightarrow P'_j(y)) \right) \Rightarrow Q(y) \right).$$

Observe that here we use θ' rather than the formula θ of Theorem 4.15, since we also want to count the "inconsistent" atomic description where all atoms are empty. Recall that, assuming θ', each simplified atomic description ψ_γ over $\mathcal{P}'$ corresponds precisely to a single maximal atomic description ψ'_γ over $\mathcal{P}' \cup \{Q\}$. We reduce the problem of counting accepting guess strings to that of counting maximal atomic descriptions over $P' \cup \{Q\}$.

We now consider cases (a) and (b) separately, beginning with the former. Fix a guess string γ. In Theorem 4.18, we constructed formulas $\varphi_w[R], \xi_\gamma \in \mathcal{L}(\Omega)$ and $\theta_w \in \mathcal{L}(\Upsilon)$ such that $\Pr^w_\infty(\varphi_w[\xi_\gamma] \mid \theta_w) = 1$ iff **M** accepts w given γ, and is 0 otherwise. Recall that the formula $\xi_\gamma(x)$ (see equation (1)) sets the ith guess bit to be γ_i by forcing the appropriate one of $B_0(x, e_i)$ and $B_1(x, e_i)$ to hold, where e_i is the unique element in the atom A_i. In Theorem 4.18, this was done directly by reference to the bits γ_i. Now, we want to derive the correct bit values from ψ_γ, which tells us that the ith bit is 1 iff $\exists z\, A'_i(z)$. The following formula ξ has precisely the desired property:

$$\xi(x) =_{\text{def}} \forall y \left(P^*(y) \Rightarrow \left(B_1(x, y) \Leftrightarrow \exists z \left(Q(z) \wedge \bigwedge_{j=1}^{t(n)} (P_j(y) \Leftrightarrow P'_j(z)) \right)\right)\right).$$

Clearly, $\psi'_\gamma \models \xi \Leftrightarrow \xi_\gamma$.

Similarly, for case (b), the formula ξ' is:

$$\xi'(x_0, x_1) =_{\text{def}} \forall y_1, \ldots, y_{t(n)} \left(\left(\bigwedge_{j=1}^{t(n)} (y_j = x_0 \vee y_j = x_1) \right) \Rightarrow \right.$$
$$\left. \left(B'_1(y_1, \ldots, y_{t(n)}) \Leftrightarrow \exists z \left(Q(z) \wedge \bigwedge_{j=1}^{t(n)} (y_j = x_1 \Leftrightarrow P'_j(z)) \right)\right)\right).$$

As in part (a), $\psi'_\gamma \models \xi' \Leftrightarrow \xi'_\gamma$.

Now, for case (a), we want to compute the asymptotic conditional probability $\Pr^w_\infty(\varphi[\xi] \mid \theta_w \wedge \theta')$. In doing this computation, we will use the observation (whose straightforward proof we leave to the reader) that if the symbols that appear in θ_2 are disjoint from those that appear in φ_1 and θ_1, then $\Pr^w_\infty(\varphi_1 \mid \theta_1 \wedge \theta_2) = \Pr^w_\infty(\varphi_1 \mid \theta_1)$. Using this observation and the fact that all maximal atomic descriptions over $\mathcal{P}' \cup \{Q\}$ are equally likely given $\theta_w \wedge \theta'$, by straightforward probabilistic reasoning we obtain:

$$\begin{aligned}\Pr^w_\infty(\varphi_w[\xi] \mid \theta_w \wedge \theta') &= \sum_{\psi'_\gamma} \Pr^w_\infty(\varphi_w[\xi] \mid \theta_w \wedge \theta' \wedge \psi'_\gamma) \cdot \Pr^w_\infty(\psi'_\gamma \mid \theta_w \wedge \theta') \\ &= \frac{1}{2^{T(n)}} \sum_{\psi'_\gamma} \Pr^w_\infty(\varphi_w[\xi] \mid \theta_w \wedge \theta' \wedge \psi'_\gamma).\end{aligned}$$

We observed before that ξ is equivalent to ξ_γ in worlds satisfying ψ'_γ, and therefore

$$\Pr^w_\infty(\varphi_w[\xi] \mid \theta_w \wedge \theta' \wedge \psi'_\gamma) = \Pr^w_\infty(\varphi_w[\xi_\gamma] \mid \theta_w \wedge \theta' \wedge \psi'_\gamma) = \Pr^w_\infty(\varphi_w[\xi_\gamma] \mid \theta_w),$$

where the second equality follows from the observation that none of the vocabulary symbols in ψ'_γ or θ' appear anywhere in $\varphi_w[\xi_\gamma]$ or in θ_w. In Theorem 4.18, we proved that $\Pr^w_\infty(\varphi_w[\xi_\gamma] \mid \theta_w)$ is equal to 1 if the ATM accepts w given γ and 0 if not. We therefore obtain that

$$\Pr^w_\infty(\varphi_w[\xi] \mid \theta_w \wedge \theta') = \frac{f(w)}{2^{T(n)}}.$$

Since both $\varphi_w[\xi]$ and $\theta_w \wedge \theta'$ are polynomial in the size of the representation of **M** and in $n = |w|$, this concludes the proof for part (a). The completion of the proof for part (b) is essentially identical. □

It remains only to investigate the problem of approximating $\Pr^w_\infty(\varphi \mid \theta)$ for this language.

THEOREM 4.20. *Fix rational numbers $0 \le r_1 \le r_2 \le 1$ such that $[r_1, r_2] \ne [0, 1]$. For $\varphi, \theta \in \mathcal{L}(\Omega)$, the problem of deciding whether $\Pr^w_\infty(\varphi \mid \theta) \in [r_1, r_2]$ is TA(EXP,LIN)-hard, even given an oracle for deciding whether the limit exists.*

Proof. For the case of $r_1 = 0$ and $r_2 < 1$, the result is an easy corollary of Theorem 4.18. We can generalize this to the case of $r_1 > 0$, using precisely the same technique as in Theorem 4.16. □

4.6. Complexity for random structures. So far in this section, we have investigated the complexity of various problems relating to the asymptotic conditional probability using the random-worlds method. We now deal with the same issues for the case of random structures. It turns out that most of our results for random worlds carry through to random structures for trivial reasons.

First, consider the issue of well-definedness. By Proposition 2.3, well-definedness is equivalent for random worlds and random structures. Therefore, all of the results obtained for random worlds carry through unchanged for random structures.

For computing or approximating the limit, Theorem 3.37 allows us to restrict attention to unary vocabularies and unary sentences φ and θ. In particular, there is no need to duplicate the results in §4.5. For the remainder of this section, we analyze the complexity of computing $\Pr^{s,\Psi}_\infty(\varphi \mid \theta)$ for $\varphi, \theta \in \mathcal{L}(\Psi)$. As before, we can assume that $\mathcal{A}^\Psi_{\varphi\wedge\theta} \subseteq \mathcal{A}^\Psi_\theta$.

The computational approach is essentially the same as that for random worlds. However, as we showed in §3.5, rather than partitioning θ into model descriptions, we can make use of the assumption that the vocabulary is unary and instead partition it into atomic descriptions ψ. That is, for $a = \alpha^\Psi(\theta)$,

$$\Pr^{s,\Psi}_\infty(\varphi \mid \theta) = \frac{1}{|\mathcal{A}^{\Psi,a}_\theta|} \cdot \sum_{\psi \in \mathcal{A}^{\Psi,a}} \Pr^w_\infty(\varphi \mid \psi) \left(= \frac{|\mathcal{A}^{\Psi,a}_{\varphi\wedge\theta}|}{|\mathcal{A}^{\Psi,a}_\theta|} \right).$$

As for random worlds, we begin with the problem of computing 0-1 probabilities. In §4.1, we showed how to extend Grandjean's algorithm to compute $\Pr^w_\infty(\varphi \mid \psi \wedge \mathcal{V})$. Fix a unary vocabulary Ψ, and suppose that $\psi \wedge \mathcal{V}$ is a model description over Ψ, with $n = \nu(\psi)$. Recall from Proposition 3.21 that $\Pr^w_\infty(\varphi \mid \psi \wedge \mathcal{V}) = \Pr^w_\infty(\varphi \mid \psi \wedge \exists x_1, \dots, x_n\, D_\mathcal{V})$. However, in the unary case it is easy to see that $\psi \wedge \exists x_1, \dots, x_n\, D_\mathcal{V}$ is equivalent to ψ. This is because the only nontrivial properties of the named elements given by $\mathcal{V}$ is which atom each of them satisfies and the equality relations between the constants, and this information is already present in the atomic description ψ.

Therefore, we conclude that $\Pr^w_\infty(\varphi \mid \psi)$ is either 0 or 1, because this is so for $\Pr^w_\infty(\varphi \mid \psi \wedge \mathcal{V})$. Now recall that if $\psi \in \mathcal{A}^\Psi_\varphi$ then ψ implies φ. In this case, clearly $\Pr^w_\infty(\varphi \mid \psi) = \Pr^{s,\Psi}_\infty(\varphi \mid \psi) = 1$. Similarly, if $\psi \notin \mathcal{A}^\Psi_\varphi$, then ψ is inconsistent with φ and $\Pr^w_\infty(\varphi \mid \psi) = \Pr^{s,\Psi}_\infty(\varphi \mid \psi) = 0$. So it follows that we can continue to use Grandjean's algorithm, as described in §4.1, to compute $\Pr^{s,\Psi}_\infty(\varphi \mid \psi)$.

THEOREM 4.21. *There exists an alternating Turing machine that takes as input a finite unary vocabulary Ψ, an atomic description ψ over Ψ, and a formula $\varphi \in \mathcal{L}(\Psi)$, and decides whether $\Pr^{s,\Psi}_\infty(\varphi \mid \psi)$ is 0 or 1. The machine uses time $O(|\Psi| 2^{|\mathcal{P}|}(\nu(\psi) + |\varphi|))$ and has at most $O((2^{|\Psi|} + \nu(\psi))^{|\varphi|})$ branches and $O(|\varphi|)$ alternations.*

As before, we can simulate the ATM deterministically.

COROLLARY 4.22. *There exists a deterministic Turing machine that takes as input a finite unary vocabulary Ψ, an atomic description ψ over Ψ, and a formula $\varphi \in \mathcal{L}(\Psi)$, and decides whether $\Pr^{w,\Psi}_\infty(\varphi \mid \psi)$ is 0 or 1. The machine uses time $2^{O(|\varphi||\Psi| \log(\nu(\psi)+1))}$ and space $O(|\varphi||\Psi| \log(\nu(\psi) + 1))$.*

We now analyze the complexity of computing $\Pr^{s,\Psi}_\infty(\varphi \mid \theta)$. We begin with the case of a fixed finite vocabulary Ψ.

THEOREM 4.23. *Fix a finite unary vocabulary Ψ with at least one predicate symbol. For $\varphi, \theta \in \mathcal{L}(\Psi)$, the problem of computing $\Pr^{s,\Psi}_{\infty}(\varphi \mid \theta)$ is PSPACE-complete. Moreover, deciding if $\Pr^{w}_{\infty}(\varphi \mid true) = 1$ is PSPACE-hard, even if $\varphi \in \mathcal{L}^{-}(\{P\})$.*

Proof. By Corollaries 3.41 and 3.42, if $\varphi, \theta \in \mathcal{L}^{-}(\{P\})$ and $P \in \Psi$, then $\Pr^{w}_{\infty}(\varphi \mid \theta) = \Pr^{s,\{P\}}_{\infty}(\varphi \mid \theta) = \Pr^{s,\Psi}_{\infty}(\varphi \mid \theta)$. Thus, the lower bound follows immediately from Theorem 4.7.

For the upper bound, we follow the same general procedure of *Compute-Pr*$_{\infty}$: generating all atomic descriptions of size $d(\theta) + |\mathcal{C}|$, and computing $\Pr^{s,\Psi}_{\infty}(\varphi \mid \theta)$. The only difference is that, rather than counting only model descriptions of the highest degree $\Delta(\psi) = (\alpha(\psi), \nu(\psi))$, we count all atomic descriptions of the highest activity count $\alpha(\psi)$. Clearly, since there are fewer atomic descriptions than model descriptions, and an atomic description has a shorter description length than a model description, the complexity of the resulting algorithm can only be lower than the corresponding complexity for random worlds. The algorithm for random structures is therefore also in PSPACE. □

Just as Theorem 4.7, Theorem 4.23 shows that even approximating the limit is hard. That is, for a fixed ϵ with $0 < \epsilon < 1$, the problem of deciding whether $\Pr^{s,\Psi}_{\infty}(\varphi \mid \theta) \in [0, 1-\epsilon]$ is PSPACE-hard even for $\varphi, \theta \in \mathcal{L}^{-}(\{P\})$. As for the case of random worlds, this lower bound cannot be generalized to arbitrary intervals $[r_1, r_2]$ unless we allow equality. In particular, for any fixed finite language, there is a fixed number of atomic descriptions of size 1, where this number depends only on the language. Therefore, there are only finitely many values that the probability $\Pr^{s,\Psi}_{\infty}(\varphi \mid \theta)$ can take for $\varphi, \theta \in \mathcal{L}^{-}(\Psi)$. However, and unlike the case for random worlds, for random structures once we have equality in the language, a single unary predicate suffices in order to have this probability assume infinitely many values.

THEOREM 4.24. *Fix a finite unary vocabulary Ψ that contains at least one unary predicate and rational numbers $0 \le r_1 \le r_2 \le 1$ such that $[r_1, r_2] \ne [0, 1]$. For $\varphi, \theta \in \mathcal{L}(\Psi)$, the problem of deciding whether $\Pr^{s,\Psi}_{\infty}(\varphi \mid \theta) \in [r_1, r_2]$ is PSPACE-hard, even given an oracle that tells us whether the limit is well defined.*

Proof. We first prove the result under the assumption that $\Psi = \{P\}$.

For the case of $r_1 = 0$ and $r_2 < 1$, the result follows trivially from Theorem 4.23. Let $r_1 = q/p > 0$. As for random worlds, we construct formulas $\varphi_{r_1}, \theta_{r_1}$ such that $\Pr^{s,\{P\}}_{\infty}(\varphi_{r_1} \mid \theta_{r_1}) = r_1$. The formula θ_{r_1} is $\exists x\, P(x) \wedge \exists^{\le p} x\, P(x)$. The formula φ_{r_1} is $\exists x\, P(x) \wedge \exists^{\le q} x\, P(x)$. Clearly, there are p atomic descriptions consistent with θ_{r_1}, among which q are also consistent with φ_{r_1}. Thus, $\Pr^{s,\{P\}}_{\infty}(\varphi_{r_1} \mid \theta_{r_1}) = q/p = r_1$.

Now, as in Theorem 4.8, let β be a QBF, and define ξ_β as in that proof. As there, $\Pr^{s,\{P\}}_{\infty}(\xi_\beta \wedge \varphi_{r_1} \mid \theta_{r_1} \wedge \exists x\, \neg P(x))$ is 0 if β is false and r_1 if it is true. Thus, by computing this probability, we can decide the truth of β, proving PSPACE-hardness in this case.

The result in the case that $\Psi \ne \{P\}$ is not immediate as it is for random worlds, since the asymptotic probability in the case of random structures may depend on the vocabulary. We define a formula θ' to be the following conjunction: for each predicate P' in $\Psi - \{P\}$, θ' contains the conjunct $\forall x P'(x)$. If Ψ contains constant symbols, let c be a fixed constant symbol in Ψ. Then θ' also contains the conjunct $\bigwedge_{P' \in \Psi} P'(c)$, and conjuncts $c = c'$ for each constant c' in Ψ. We leave it to the reader to check that for any formulas $\varphi, \theta \in \mathcal{L}(\{P\})$, $\Pr^{s,\{P\}}_{\infty}(\varphi \mid \theta) = \Pr^{s,\Psi}_{\infty}(\varphi \mid \theta \wedge \theta')$. □

For the case of a finite vocabulary and a bound on the quantifier depth, precisely the same argument as that given for Theorem 4.9 allows us to show the following.

THEOREM 4.25. *Fix $d \ge 0$. For $\varphi, \theta \in \mathcal{L}(\Psi)$ such that $d(\varphi), d(\theta) \le d$, we can effectively construct a linear time algorithm that decides if $\Pr^{s,\Psi}_{\infty}(\varphi \mid \theta)$ is well defined and computes it if it is.*

We now drop the assumption that we have a fixed finite vocabulary. As we previously discussed, there are at least two distinct interpretations for asymptotic conditional probabilities in this case. One interpretation of "infinite vocabulary" views Ω as a potential or background vocabulary, so that every problem instance includes as part of its input the actual finite subvocabulary that is of interest. So although this subvocabulary is finite, there is no bound on its possible size. The alternative is to interpret infinite vocabularies more literally, using the limit process explained in §2.3. As we mentioned, for random worlds the two interpretations are equivalent. However, this is not the case for random structures, where the two interpretations may give different answers. In fact, from Corollary 2.9, it follows that the random-structures method reduces to the random-worlds method under the second interpretation. Thus, the complexity results are the same for random worlds and random structures under this interpretation. As we already observed, even under the first interpretation, the random-structures method reduces to the random-worlds method if there is a binary predicate in the language. It therefore remains to prove the complexity results for random structures only for the first interpretation, where the vocabulary is considered to be part of the input, under the assumption that the language is unary. As Example 2.4 shows, in this case, the random-worlds method may give answers different from those given by the random-structures method. Nevertheless, as we now show, the same complexity bounds hold for both random worlds and random structures.

As for the case of the finite vocabulary, the lower bounds for computing the probability (Theorem 4.15) and for approximating it (Theorem 4.16) only use formulas in $\mathcal{L}^-(\mathcal{P})$ for some $\mathcal{P} \subset \mathcal{Q}$. Therefore, by Corollaries 3.41 and 3.42, the lower bounds hold unchanged for random structures.

THEOREM 4.26. *For $\Psi \subset \Upsilon$ and $\varphi, \theta \in \mathcal{L}^-(\mathcal{P})$ of depth at least 2, the problem of computing $\Pr_\infty^{s,\Psi}(\varphi \mid \theta)$ is #EXP-hard, even given an oracle for deciding whether the limit exists.*

THEOREM 4.27. *Fix rational numbers $0 \leq r_1 \leq r_2 \leq 1$ such that $[r_1, r_2] \neq [0, 1]$. For $\Psi \subseteq \Omega$ and $\varphi, \theta \in \mathcal{L}^-(\mathcal{P})$ of depth 2, the problem of deciding whether $\Pr_\infty^{s,\Psi}(\varphi \mid \theta) \in [r_1, r_2]$ is both NEXPTIME-hard and co-NEXPTIME-hard, even given an oracle for deciding whether the limit exists.*

It remains only to prove the #EXP upper bound for computing the asymptotic probability.

THEOREM 4.28. *For $\Psi \subseteq \Omega$ and $\varphi, \theta \in \mathcal{L}(\Psi)$, the problem of computing $\Pr_\infty^{s,\Psi}(\varphi \mid \theta)$ is #EXP-easy.*

Proof. We again follow the outline of the proof for the case of random worlds. Recall that in the proof of Theorem 4.13 we construct a Turing machine such that the number of accepting paths of **M** encodes, for each degree δ, $count^\delta(\varphi)$ and $count^\delta(\theta)$. From this encoding we could deduce the maximum degree, and calculate the asymptotic conditional probability. This was accomplished by guessing a model description $\psi \wedge \mathcal{V}$, and branching sufficiently often, according to $\Delta(\psi)$, so that the different counts in the output are guaranteed to be separated. The construction for random structures is identical, except that we guess atomic descriptions ψ rather than model descriptions, and branch according to $\alpha(\psi)$ rather than according to $\Delta(\psi)$. Again, since there are fewer atomic descriptions than model descriptions, and the representation of atomic descriptions is shorter, the resulting Turing machine is less complex, and therefore also in #EXP. □

5. Conclusions. In this paper and [23], we have carried out a rather exhaustive study of complexity issues for two principled methods for computing degrees of belief: the random-worlds method and the random-structures method. These are clearly not the only methods that one can imagine for this purpose. However, as discussed in [2] and [22], both methods are often successful at giving answers that are intuitively plausible and which agree with well-known desiderata. We believe this success justifies a careful examination of complexity issues.

Here we have focused on the case where the formula we are conditioning on is a unary first-order formula. As we mentioned in the introduction, in many applications we want to move beyond first order and also allow for statistical knowledge. Both methods continue to make sense in this case. Furthermore, as shown in [33], [3], and [27], for a unary language we can often calculate asymptotic probabilities in the random-worlds method, using a combination of the techniques in this paper and the principle of maximum entropy. Since a lot is already known about computing maximum entropy (for example, [7], [9], [19]), this combination may lead to efficient algorithms for some practical problems.

Acknowledgments. We would like to thank Fahiem Bacchus, with whom we started working on this general area of research. We would also like to thank Ron Fagin and Moshe Vardi for useful comments on a previous draft of this paper. Ron's comments encouraged us to discuss both the random-structures and the random-worlds methods in this paper. We would also like to thank Moshe for pointing out the reference [31]. Finally, we would like to thank Andrew Granville for help with some of the details of Theorem 4.8.

REFERENCES

[1] W. Ackermann, *Solvable cases of the decision problem*, North–Holland, Amsterdam, 1954.

[2] F. Bacchus, A. J. Grove, J. Y. Halpern, and D. Koller, *From statistics to belief*, in Proc. National Conference on Artificial Intelligence (AAAI '92), AIAA Press/MIT Press, New York, 1992, pp. 602–608.

[3] ———, *From statistical knowledge bases to degrees of belief*, Tech. Report 9855, IBM Almaden Research Center, San Jose, CA, 1994; Artif. Intell., to appear. Also available by anonymous ftp from logos.uwaterloo.ca/pub/bacchus or via WWW at http://logos.uwaterloo.ca. A preliminary version of this work appeared in Proc. International Joint Conference on Artificial Intelligence (IJCAI '93), Chambéry, France, 1993, pp. 563–569.

[4] A. Blass, Y. Gurevich, and D. Kozen, *A zero–one law for logic with a fixed point operator*, Inform. and Control, 67 (1985), pp. 70–90.

[5] A. K. Chandra, D. Kozen, and L. J. Stockmeyer, *Alternation*, J. Assoc. Comput. Mach., 28 (1981), pp. 114–133.

[6] A. K. Chandra and P. M. Merlin, *Optimal implementation of conjunctive queries in relational databases*, in Proc. 9th ACM Symp. on Theory of Computing, Boulder, CO, 1977, pp. 77–90.

[7] P. C. Cheeseman, *A method of computing generalized Bayesian probability values for expert systems*, in Proc. Eighth International Joint Conference on Artificial Intelligence (IJCAI '83), Karlsruhe, Germany, 1983, pp. 198–202.

[8] K. Compton, 0-1 *laws in logic and combinatorics*, in Proc. 1987 NATO Adv. Study Inst. on algorithms and order, I. Rival, ed., Reidel, Dordrecht, the Netherlands, 1988, pp. 353–383.

[9] W. E. Deming and F. F. Stephan, *On a least squares adjustment of a sampledfrequency table when the expected marginal totals are known*, Ann. Math. Stat., 11 (1940), pp. 427–444.

[10] K. G. Denbigh and J. S. Denbigh, *Entropy in Relation to Incomplete Knowledge*, Cambridge University Press, Cambridge, UK, 1985.

[11] B. Dreben and W. D. Goldfarb, *The Decision Problem: Solvable Classes of Quantificational Formulas*, Addison–Wesley, Reading, MA, 1979.

[12] H. B. Enderton, *A Mathematical Introduction to Logic*, Academic Press, New York, 1972.

[13] R. Fagin, *Probabilities on finite models*, J. Symbol. Logic, 41 (1976), pp. 50–58.

[14] ———, *The number of finite relational structures*, Discrete Math., 19 (1977), pp. 17–21.

[15] H. Gaifman, *Probability models and the completeness theorem*, in Internat. Congress of Logic Methodology and Philosophy of Science, 1960, pp. 77–78. This is the abstract of which [16] is the full paper.

[16] ———, *Concerning measures in first order calculi*, Israel J. Math., 2 (1964), pp. 1–18.

[17] M. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-completeness*, W. Freeman and Co., San Francisco, CA, 1979.

[18] Y. V. Glebskiĭ, D. I. Kogan, M. I. Lĭogon'kĭĭ, and V. A. Talanov, *Range and degree of realizability of formulas in the restricted predicate calculus*, Kibernetika, 2 (1969), pp. 17–28.

[19] S. A. Goldman, *Efficient methods for calculating maximum entropy distributions*, Master's thesis, EECS Department, MIT, Cambridge, MA, 1987.

[20] E. Grandjean, *Complexity of the first-order theory of almost all structures*, Inform. and Control, 52 (1983), pp. 180–204.

[21] A. J. Grove, J. Y. Halpern, and D. Koller, *Asymptotic conditional probabilities for first-order logic*, in Proc. 24th ACM Symp. on Theory of Computing, 1992, pp. 294–305.

[22] A. J. Grove, J. Y. Halpern, and D. Koller, *Random worlds and maximum entropy*, J.A.I. Res., 2 (1994), pp. 33–38.

[23] ———, *Asymptotic conditional probabilities: The non-unary case*, Research Report RJ 9564, IBM Almaden Research Center, San Jose, CA, 1993; J. Symbol. Logic, to appear.

[24] I. Hacking, *The Emergence of Probability*, Cambridge University Press, Cambridge, UK, 1975.

[25] E. T. Jaynes, *Where do we stand on maximum entropy?*, in The Maximum Entropy Formalism, R. D. Levine and M. Tribus, eds., MIT Press, Cambridge, MA, 1978, pp. 15–118.

[26] J. M. Keynes, *A Treatise on Probability*, Macmillan, London, 1921.

[27] D. Koller, *From Knowledge to Belief*, Ph.D. thesis, Dept. of Computer Science, Stanford University, 1994.

[28] J. v. Kries, *Die Principien der Wahrscheinlichkeitsrechnung und Rational Expectation*, Freiburg, 1886.

[29] H. R. Lewis, *Unsolvable Classes of Quantificational Formulas*, Addison-Wesley, New York, 1979.

[30] ———, *Complexity results for classes of quantificational formulas*, J. Comput. System Sci., 21 (1980), pp. 317–353.

[31] M. I. Līogon'kĭĭ, *On the conditional satisfiability ratio of logical formulas*, Math. Notes Acad. USSR, 6 (1969), pp. 856–861.

[32] J. Lynch, *Almost sure theories*, Ann. Math. Logic, 18 (1980), pp. 91–135.

[33] J. B. Paris and A. Vencovska, *On the applicability of maximum entropy to inexact reasoning*, Internat. J. Approx. Reasoning, 3 (1989), pp. 1–34.

[34] S. J. Provan and M. O. Ball, *The complexity of counting cuts and of computing the probability that a graph is connected*, SIAM J. Comput., 12 (1983), pp. 777–788.

[35] D. Roth, *On the hardness of approximate reasoning*, in Proc. Thirteenth International Joint Conference on Artificial Intelligence (IJCAI '93), Chambéry, France, 1993, pp. 613–618.

[36] G. Shafer, *Personal communication*, 1993.

[37] L. J. Stockmeyer, *The polynomial-time hierarchy*, Theoret. Comput. Sci., 3 (1977), pp. 1–22.

[38] L. G. Valiant, *The complexity of computing the permanent*, Theoret. Comput. Sci., 8 (1979), pp. 189–201.

[39] ———, *The complexity of enumeration and reliability problems*, SIAM J. Comput., 8 (1979), pp. 410–421.

[40] R. L. Vaught, *Applications of the Lowenheim-Skolem-Tarski theorem to problems of completeness and decidability*, Indag. Math., 16 (1954), pp. 467–472.

SIAM J. COMPUT.
Vol. 25, No. 1, pp. 52-82, February 1996

A FAST DERANDOMIZATION SCHEME AND ITS APPLICATIONS*

YIJIE HAN†

Abstract. This paper presents a fast derandomization scheme for the PROFIT/COST problem. Through the applications of this scheme the time complexity of $O(\log^2 n \log\log n)$ for the $\Delta + 1$ vertex-coloring problem using $O((m+n)/\log\log n)$ processors on the concurrent read exclusive write parallel random-access machine (CREW PRAM), the time complexity of $O(\log^{2.5} n)$ for the maximal independent set problem using $O((m+n)/\log^{1.5} n)$ processors on the CREW PRAM and the time complexity of $O(\log^{2.5} n)$ for the maximal matching problem using $O((m+n)/\log^{0.5} n)$ processors on the exclusive read exclusive write (EREW) PRAM are shown.

Key words. derandomization, parallel algorithms, graph algorithms, graph coloring, maximal independent set, maximal matching

AMS subject classifications. 05C70, 05C85, 68Q20, 68Q22, 68Q25, 68R10

1. Introduction.

Recent progress in derandomization has resulted in several efficient sequential and parallel algorithms [ABI], [BR], [BRS], [H], [HI], [KW], [L1], [L2], [L3], [MNN], [PSZ], [Rag], [Sp]. The essence of the technique of derandomization lies in the design of a sample space that is easy to search, in the probabilistic analysis showing that the expectation of a desired random variable is no less than demanded, and in the search technique which ultimately returns a good sample point. Raghavan [Rag] and Spencer [Sp] showed how to search an exponential-size sample space to obtain polynomial-time sequential algorithms. Their technique cannot be applied directly to obtain efficient parallel algorithms through derandomization. Alon et al. [ABI], Karp and Wigderson [KW], and Luby [L1], [L2], [L3] developed schemes using $O(n)$ random variables with limited independence on a small sample space and thus obtained efficient parallel algorithms through derandomization. Berger and Rompel [BR] and Motwani et al. [MNN] presented novel designs in which $(\log^c n)$-wise independent random variables are used in randomized algorithms and then $\mathcal{NC}$ [Co] algorithms are obtained through the derandomization of these randomized algorithms.

To obtain efficient parallel algorithms, i.e., algorithms using no more than a linear number of processors and running in polylog time, Luby [L2], [L3] outlined an elegant framework in which pairwise independent random variables are designed on a sample space that facilitates binary search. His framework [L2], [L3] consists of a derandomization scheme for the bit-pairs PROFIT/COST problem and the general-pairs PROFIT/COST problem, and applications of the scheme to three problems: the $\Delta + 1$ vertex-coloring problem, the maximal independent set problem, and the maximal matching problem. By applying his derandomization scheme, he obtained a linear processor concurrent read exclusive write (CREW) algorithm for the $\Delta + 1$ vertex-coloring problem with time complexity $O(\log^3 n \log\log n)$. Although he put the three problems in a very nice setting, his derandomization scheme is not efficient enough to improve on parallel algorithms for the maximal independent set problem and the maximal matching problem obtained through ad hoc designs [GS2], [IS]. To illustrate his ideas, Luby gave linear processor algorithms for the maximal independent set problem and the maximal matching problem with time complexity $O(\log^5 n)$ through derandomization [L3].

Recently, Han and Igarashi [HI] gave a fast derandomization scheme for the bit-pairs PROFIT/COST problem. Their scheme yields a fast CREW parallel algorithm for

*Received by the editors January 11, 1991; accepted for publication (in revised form) July 22, 1994. Part of this paper appeared in *Proc.* 1991 *Workshop on Algorithms and Data Structures*, Carleton University, Ottawa, ON, Canada, Lecture Notes in Comput. Sci. 519, Springer-Verlag, Berlin, New York, 1991, pp. 177-188.

†Department of Computer Science, University of Kentucky, Lexington, KY 40506. Current address: Electronic Data Systems, Inc., 37350 Ecorse Road, Romulus, MI 48174.

the bit-pairs PROFIT/COST problem with time complexity $O(\log n)$ using a linear number of processors. Han then showed [H] how to obtain an exclusive read exclusive write (EREW) algorithm with the same time and processor complexities. Their result improves the time complexity of Luby's $\Delta + 1$ vertex-coloring algorithm to $O(\log^3 n)$. The most interesting feature in Han and Igarashi's scheme [H], [HI] is the design of a sample space which allows redundancy and mutual independence to be exploited.

In this paper, we give a new scheme to speed up the derandomization process of the general-pairs PROFIT/COST problem. This scheme allows several bit-pairs PROFIT/COST problems in one general-pairs PROFIT/COST problem to be solved in one pass. We note that our scheme cannot be constructed efficiently under the setting of previous derandomization schemes [L2], [L3] because it would require more than a linear number of processors. The power of our derandomization scheme allows us to improve the time complexity for the $\Delta + 1$ vertex-coloring problem and to obtain new efficient parallel algorithms for the maximal independent set problem and the maximal matching problem.

A substantial amount of effort has been put into the search for efficient parallel algorithms for these three problems. There are important special cases where optimal parallel algorithms are known. Hagerup et al. [HCD] have an optimal parallel algorithm for the 5-coloring of planar graphs which implies an optimal parallel algorithm for the maximal independent set problem for planar graphs. Significant progress has also been made on the three problems for graphs [ABI], [GS1], [GS2], [HI], [IS], [KW], [L1], [L2], [L3]. In this paper, we only study these three problems on graphs.

Luby obtained through derandomization a CREW algorithm for the $\Delta + 1$ vertex-coloring problem with time complexity $O(\log^3 n \log\log n)$ using a linear number of processors. Han and Igarashi's work [HI] improves the time complexity for the $\Delta + 1$ vertex-coloring problem to $O(\log^3 n)$. In this paper, we improve the time complexity for the $\Delta + 1$ vertex-coloring problem to $O(\log^2 n \log\log n)$ using $O((m+n)/\log\log n)$ processors on the CREW parallel random-access machine (PRAM) [FW]. For the maximal independent set problem, the first $\mathcal{NC}$ algorithm, which was obtained by Karp and Wigderson [KW], has time complexity $O(\log^4 n)$ using $O(n^3/\log^3 n)$ processors on the EREW PRAM. Their result has since been improved to time $O(\log^2 n)$ using $O(mn^2)$ processors on the EREW PRAM by Luby [L1] and to time $O(\log^3 n)$ using $O((m+n)/\log n)$ processors on the EREW PRAM by Goldberg and Spencer [GS1], [GS2]. In this paper, we obtain an EREW algorithm with time complexity $O(\log^{2.5} n)$ using $O((m+n)/\log^{0.5} n)$ processors. We are able to achieve the same time complexity using $O((m+n)/\log^{1.5} n)$ processors on the CREW PRAM. We also obtain a CREW algorithm with time complexity $O(\log^2 n)$ using $O(n^{2.376})$ processors. For the maximal matching problem, Israeli and Shiloach's concurrent read concurrent write (CRCW) algorithm [IS] has time complexity $O(\log^3 n)$ using $O(m+n)$ processors. In this paper, we give an EREW algorithm with time complexity $O(\log^{2.5} n)$ using $O((m+n)/\log^{0.5} n)$ processors.

Since our algorithms are obtained through derandomization, they are derived at a loss of efficiency from the original randomized algorithms.

Our approach to the three problems follows that of Luby's [L2], [L3]. Our results are obtained through the novel design of sample spaces which follows Han and Igarashi's work [H], [HI], the formulation of our fast derandomization process, and the adaptive applications of the fast derandomization techniques to the three problems.

We have outlined here only the main results achieved. There are ramifications of our results which we will mention in the remaining sections of the paper.

2. Preliminaries. The *bit-pairs PROFIT/COST* (BPC) and the *general-pairs PROFIT/COST* (GPC) problems as formulated by Luby [L2] can be described as follows.

Let $\vec{x} =< x_i \in \{0, 1\}^q : i = 0, \ldots, n-1 >$. Each point $\vec{x}$ out of the 2^{nq} points is assigned probability $1/2^{nq}$. Given function $B(\vec{x}) = \sum_{i,j} f_{i,j}(x_i, x_j)$, where $f_{i,j}$ is defined as a function $\{0, 1\}^q \times \{0, 1\}^q \rightarrow \mathcal{R}$. The GPC problem is to find a good point $\vec{y}$ such that $B(\vec{y}) \geq E[B(\vec{x})]$. B is called the general-pairs BENEFIT function and the $f_{i,j}$'s are called the GPC functions. When $q = 1$, the GPC problem is called a BPC problem and the $f_{i,j}$'s are called the BPC functions.

The size m of the problem is the number of nontrivial PROFIT/COST functions present in the input.

Let $G = (V, E)$ be a graph with $|V| = n$ and $|E| = m$. The degree of $v \in V$ is denoted by $d(v)$. Let $\Delta = \max\{d(v) | v \in V\}$. The output of the $\Delta + 1$ vertex coloring is $color(v) \in \{1, \ldots, \Delta + 1\}$ for all $v \in V$ such that if $(i, j) \in E$, then $color(i) \neq color(j)$. $I \subseteq V$ is an independent set if for $v, w \in I, v \neq w$, $(v, w) \notin E$. I is a maximal independent set if I is not a proper subset of any other independent set. $M \subseteq E$ is a matching set if no two edges in M have a common vertex. M is maximal if it is not a proper subset of any other matching set.

Han and Igarashi [HI] have formulated the BPC problem as a tree-contraction problem [MR]. Without loss of generality, assume that n is a power of 2. n 0/1-valued uniformly distributed mutually independent random variables r_i, $0 \leq i < n$, are used. A *random-variable tree* T is built for $\vec{x}$. T is a complete binary tree with n leaves plus a node which is the parent of the root of the complete binary tree (thus there are n interior nodes in T and the root of T has only one child). The n variables x_i, $0 \leq i < n$, are associated with n leaves of T, and the n random variables r_i, $0 \leq i < n$, are associated with the interior nodes of T. The n leaves of T are numbered from 0 to $n - 1$. Variable x_i is associated with leaf i.

Variables x_i, $0 \leq i < n$, are chosen randomly as follows. Let $\vec{r} =< r_i : i = 0, \ldots, n - 1 >$ and let $r_{i_0}, r_{i_1}, \ldots, r_{i_{\log n}}$ be the random variables on the path from leaf i to the root of T. Random variable x_i is defined to be $x_i(\vec{r}) = (\sum_{j=0}^{\log n - 1} i_j \cdot r_{i_j} + r_{i_{\log n}}) \bmod 2$, where i_j is the jth bit of i starting with the least significant bit. It can be verified [H] that random variables x_i, $0 \leq i < n$, are uniformly distributed mutually independent random variables.

Due to the linearity of expectation and pairwise independence of random variables in $\vec{x}$, $E[B(\vec{x})] = \sum_{i,j} E[f_{i,j}(x_i, x_j)] = \sum_{i,j} E[f_{i,j}(x_i(\vec{r}), x_j(\vec{r}))] = E[B(\vec{x}\ (\vec{r}))]$. The problem now is to find a sample point $\vec{r}$ such that $B(\vec{r}) \geq E[B] = \frac{1}{4} \sum_{i,j} (f_{i,j}(0, 0) + f_{i,j}(0, 1) + f_{i,j}(1, 0) + f_{i,j}(1, 1))$.

Han and Igarashi's algorithm [HI] fixes random variables r_i (setting their values to 0's and 1's) one level in a step starting from the level next to the leaves (level 0) and going upward on the tree T until level $\log n$. Since there are $\log n + 1$ interior levels in T, all random variables will be fixed in $\log n + 1$ steps.

Let random variable r_i at level 0 be the parent of the random variables x_i and $x_{i\#0}$ in the random variable tree, where $i\#j$ is a number obtained by complementing the jth bit of i. r_i will be fixed as follows. Compute $f_0 = E[f_{i,i\#0} + f_{i\#0,i} | r_i = 0] = (f_{i,i\#0}(0, 0) + f_{i,i\#0}(1, 1) + f_{i\#0,i}(0, 0) + f_{i\#0,i}(1, 1))/2$ and $f_1 = E[f_{i,i\#0} + f_{i\#0,i} | r_i = 1] = (f_{i,i\#0}(0, 1) + f_{i,i\#0}(1, 0) + f_{i\#0,i}(0, 1) + f_{i\#0,i}(1, 0))/2$. If $f_0 \geq f_1$, then set r_i to 0 else set r_i to 1. All random variables at level 0 will be fixed in parallel in constant time using n processors. This results in a smaller space with higher expectation for B. Therefore, this smaller space contains a good point.

If r_i is set to 0, then $x_i = x_{i\#0}$; if r_i is set to 1, then $x_i = 1 - x_{i\#0}$. Therefore, after r_i is fixed, x_i and $x_{i\#0}$ can be combined. The n random variables x_i, $0 \leq i < n$, can be reduced to $n/2$ random variables. PROFIT/COST functions $f_{i,j}$, $f_{i\#0,j}$, $f_{i,j\#0}$, and $f_{i\#0,j\#0}$ can also be combined into one function. It can be checked that the combining can be done in constant time using a linear number of processors.

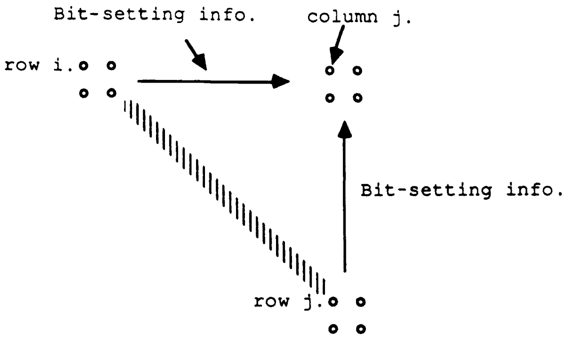


FIG. 1.

During the combining process, variables x_i and $x_{i\#0}$ are combined into a new variable $x^{(1)}_{\lfloor i/2 \rfloor}$, and functions $f_{i,j}$, $f_{i\#0,j}$, $f_{i,j\#0}$, and $f_{i\#0,j\#0}$ are combined into a new function $f^{(1)}_{\lfloor i/2 \rfloor, \lfloor j/2 \rfloor}$. After combining, a new function $B^{(1)}$ is formed which has the same form as B but has only $n/2$ variables. As we stated above, $E[B^{(1)}] \geq E[B]$.

What we have explained above is the first step of the algorithm in [HI]. This step takes constant time using a linear number of processors. After k steps, the random variables at levels 0 to $k-1$ in the random-variable tree are fixed, the n random variables $\{x_0, x_1, \ldots, x_{n-1}\}$ are reduced to $n/2^k$ random variables $\{x^{(k)}_0, x^{(k)}_1, \ldots, x^{(k)}_{n/2^k-1}\}$, and functions $f_{i,j}, i, j \in \{0, 1, \ldots, n-1\}$, have been combined into $f^{(k)}_{i,j}, i, j \in \{0, 1, \ldots, n/2^k - 1\}$.

After $\log n$ steps, $B^{(\log n)} = f^{(\log n)}_{0,0}(x^{(\log n)}_0, x^{(\log n)}_0)$. The bit at the root of the random-variable tree is now set to 0 if $f^{(\log n)}_{0,0}(0, 0) \geq f^{(\log n)}_{0,0}(1, 1)$, and 1 otherwise. Thus Han and Igarashi's algorithm [HI] solves the BPC problem in $O(\log n)$ time with a linear number of processors.

Let $n = 2^k$ and A be an $n \times n$ array. Elements $A[i, j]$, $A[i, j\#0]$, $A[i\#0, j]$, and $A[i\#0, j\#0]$ form a *gang*, which is denoted by $g_A[\lfloor i/2 \rfloor, \lfloor j/2 \rfloor]$. All gangs in A form array g_A.

When visualized on a two-dimensional array A (as shown in Fig. 1), a stage of Han and Igarashi's algorithm can be interpreted as follows. Let function $f_{i,j}$ be stored at $A[i, j]$. Setting the random variables at level 0 of the random-variable tree is done by examining the PROFIT/COST functions in the diagonal gang of A. Function $f_{i,j}$ then gets the bit-setting information from $g_A[\lfloor i/2 \rfloor, \lfloor i/2 \rfloor]$ and $g_A[\lfloor j/2 \rfloor, \lfloor j/2 \rfloor]$ to determine how it is to be combined with other functions in $g_A[\lfloor i/2 \rfloor, \lfloor j/2 \rfloor]$.

A *derandomization tree D* can be built which reflects the way the BPC functions are combined. D is of the following form. The input BPC functions are stored at the leaves, $f_{i,j}$ is stored in $A_0[i, j]$. A node $A_l[i, j]$ at level $l > 0$ is defined if there exist input functions in the range $A_0[u, v]$, $i * 2^l \leq u < (i+1) * 2^l$, $j * 2^l \leq v < (j+1) * 2^l$. A derandomization tree is shown in Fig. 2.

Tree D can be constructed [H], [HI] by first sorting the input into the file-major indexing and then building the tree bottom-up. The derandomization tree helps to reduce the space requirement for the BPC problem. Han and Igarashi's algorithm [HI] has time complexity $O(\log n)$ using a linear number of processors.

The algorithm given by Han and Igarashi [HI] is a CREW algorithm. Recently, Han [H] has given an EREW algorithm for the BPC problem with time complexity $O(\log n)$ using a linear number of processors.

We now discuss some variations of the above algorithm to be used in §§5 and 6.

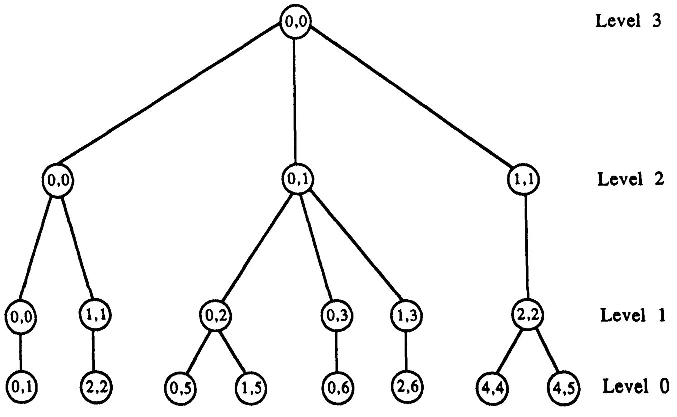


FIG. 2. *A derandomization tree. Pairs in the circles and the subscripts of PROFIT/COST problems.*

In some applications, the BPC functions cannot be combined in order to obtain an efficient algorithm. If the functions are not combined, then there could be several BPC functions $f_1(x_{i_1}, x_{j_1})$, $f_2(x_{i_2}, x_{j_2}), \ldots, f_l(x_{i_k}, x_{j_k})$ associated with an interior node r in the random-variable tree, where x_{i_t}, x_{j_t} are leaves in the subtree rooted at r. If r is not the root of the whole random-variable tree, then one of x_{i_t}, x_{j_t} is in the left subtree of r and the other in the right subtree of r, $1 \leq t \leq k$.

Let x_i be a leaf of a random-variable subtree T. Let the random variables on the path from x_i to the root r of T be set to $a_0, a_1, \ldots, a_l$. We define $\Psi(x_i, r) = \sum_{j=0}^{l}(a_j \cdot i_j) \bmod 2$. This function resembles the Ψ function defined by Luby [L2], [L3].

In fixing r, we tentatively set r to 0 and 1, respectively, and evaluate $f_t(\Psi(x_{i_t}, r)$, $\Psi(x_{j_t}, r)) + f_t(\Psi(x_{i_t}, r) \oplus 1, \Psi(x_{j_t}, r) \oplus 1)$, $1 \leq t \leq k$, where $\oplus$ is the exclusive-or function. We then get the sum of these functions and compare the sum for $r = 0$ with the sum for $r = 1$ to decide whether r should be set to 0 or 1. It takes $O(\log n)$ time to get the sum because there are at most $O(n^2)$ functions.

We note that $\Psi(x_i, r)$ can be evaluated progressively as the derandomization process proceeds, i.e., $\Psi(x_i, r) = (\Psi(x_i, r') + tr) \bmod 2$, where r is the parent of r' and t is the bit of i corresponding to r.

Thus if the functions are not combined in the derandomization process, a BPC problem requires $O(\log^2 n)$ time to solve.

In our applications we also use a combination of Luby's technique [L2] and Han and Igarashi's technique [H], [HI] for solving a BPC problem. The random-variable tree used in Luby's algorithm degenerates to a chain of length $\log n + 1$ plus n leaves. Therefore there are $\log n + 1$ random variables $r_0, r_1, \ldots, r_{\log n}$ associated with the interior nodes in the tree. x_i is chosen randomly by the formula $x_i = (\sum_{j=0}^{\log n - 1} i_j \cdot r_j + r_{\log n}) \bmod 2$. It can be shown [L2] that x_i's are pairwise independent random variables. In Luby's algorithm, the random variables in the random-variable tree are also fixed one level at a time. His algorithm takes $O(\log n)$ time to fix one level resulting in time complexity $O(\log^2 n)$. We stress that the random-variable tree used in Luby's algorithm has only $\log n + 1$ random variables. Thus the sample space contains only $2n$ sample points, while the sample space used in Han and Igarashi's algorithm [H], [HI] has 2^n sample points.

Combining Luby's technique and Han and Igarashi's technique [H], [HI], we could solve a BPC problem by using a random-variable tree T as shown in Fig. 3(c). T has $S = \lceil(\log n + 1)/a\rceil$ blocks, where a is a parameter. Block s contains levels as to $a(s+1) - 1$ of T. Block 0 has $C_0 = \lceil n/2^a \rceil$ chains. Block s has $C_s = \lceil C_{s-1}/2^a \rceil$ chains. Each chain

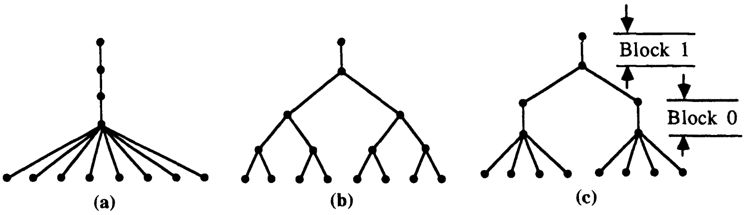


FIG. 3. (a) *Luby's tree.* (b) *Han and Igarashi's tree.* (c) *A tree of combination.*

in block s has length a running from level as to level $a(s+1)-1$. A node at level as in a chain (except the one in the last chain) has 2^a children at level $as-1$. Block $S-1$ has only one chain of length $\log n+1-a(S-1)$. There is a random bit r at each interior node of T and random variable x_i is associated with the ith leaf of T. x_i is chosen randomly as $x_i(\vec{r}) = (\sum_{j=0}^{\log n-1} i_j \cdot r_{i_j} + r_{i_{\log n}}) \bmod 2$, where $r_{i_0}, r_{i_1}, \ldots, r_{i_{\log n}}$ are the random variables on the path from leaf i to the root of T. It is straightforward to show that the x_i's are uniformly distributed pairwise independent random variables.

Different random variable trees are shown in Fig. 3.

3. A scheme for the GPC problem. In this section, we present a scheme to speed up the derandomization process for the GPC problem.

In [L2] and [L3], Luby presented the following derandomization scheme for solving the GPC problem.

Let $\vec{y} = \langle y_i \in \{0,1\}^p : i = 0, 1, \ldots, n-1\rangle$. Let $\vec{x_u}$, $p \le u < q$, be totally independent random bit strings, each of length n. Let $\vec{z}$ be a vector of n bits. We write the BENEFIT function $B(x_0, x_1, \ldots, x_{n-1})$, where each x_i is a variable containing q bits, as $B(\vec{x_{q-1}} \cdots \vec{x_{p+1}}\vec{x_p}\vec{y})$, where $\vec{y}$ contains the least significant p bits of all variables and $\vec{x_i}$ contains the ith bits of all variables. Define $TB(\vec{y}) = E[B(\vec{x_{q-1}} \cdots \vec{x_{p+1}}\vec{x_p}\vec{y})]$.Then $E[TB(\vec{x_p}\vec{y})] = E[E[B(\vec{x_{q-1}} \cdots \vec{x_{p+1}}\vec{z}\vec{y})| \vec{z} = \vec{x_p}]] = E[B(\vec{x_{q-1}} \cdots \vec{x_p}\vec{y})] = TB(\vec{y})$. That is, $TB(\vec{y})$ can also be obtained by first computing $TB(\vec{x_p}\vec{y})$, and $TB(\vec{y}) = E[TB(\vec{x_p}\vec{y})]$ $= \sum_{\vec{z}} TB(\vec{z}\vec{y})Pr(\vec{x_p} = \vec{z} \mid \vec{x_{p-1}} \cdots \vec{x_0} = \vec{y})$. Thus there exists a $\vec{z}$ such that $TB(\vec{z}\vec{y}) \ge TB(\vec{y})$. After $TB(\vec{x_p}\vec{y})$ are evaluated for all values of $\vec{x_p}$, the problem of finding such a $\vec{z}$ is a BPC problem. Because in the GPC problem function B is the sum of GPC functions, each depending on at most two variables, pairwise independent random variables can be used for bits in each $\vec{x_u}$, $p+1 \le u < q$. Luby's algorithm for the GPC problem then uses his algorithm for the BPC problem to find a $\vec{z}$ satisfying $TB(\vec{z}\vec{y}) \ge E[TB(\vec{x_p}\vec{y})] = TB(\vec{y})$, thus fixing the random bits in $\vec{x_p}$.

Luby's solution [L2], [L3] to the GPC problem can be interpreted as follows: it solves the GPC problem by solving q BPC problems, one for each $\vec{x_u}$. These BPC problems are solved sequentially. After the BPC problems for $\vec{x_u}$, $0 \le u < v$, are solved, BPC functions $f_{i_v,j_v}(x_{i_v}, x_{j_v})$ are evaluated based on the setting of bits x_{i_u}, x_{j_u}, $0 \le u < v$. Suppose x_{i_u} is set to y_{i_u} and x_{j_u} is set to y_{j_u}, $0 \le u < v$. $f_{i_v,j_v}(x_{i_v}, x_{j_v})$ is evaluated as $f_{i_v,j_v}(y_{i_v}, y_{j_v}) = E[f_{i,j}(x_i, x_j)|x_{i_0} = y_{i_0}, x_{j_0} = y_{j_0}, \ldots, x_{i_{v-1}} = y_{i_{v-1}}, x_{j_{v-1}} = y_{j_{v-1}}, x_{i_v} = y_{i_v}, x_{j_v} = y_{j_v}]$, $y_{i_v}, y_{j_v} = 0, 1$. After BPC functions $f_{i_v,j_v}(x_{i_v}, x_{j_v})$ have been obtained and stored in a table, the BPC algorithm is invoked to fix $\vec{x_v}$.

If we are to solve several BPC problems in a GPC problem simultaneously, we must have BPC functions $f_{i_v,j_v}(x_{i_v}, x_{j_v})$ before the setting of the bits x_{i_u} and x_{j_u}, $0 \le u < v$. Since there are a total of $2v$ bits, we could try out all possible 4^v bit patterns. For each bit pattern we have a distinct function $f_{i_v,j_v}(x_{i_v}, x_{j_v})$. If $q = O(\log n)$, we need only a polynomial number of processors to work on all these functions.

If we use Luby's random-variable tree for each BPC problem, then there are $\log n + 1$ random variables and $2n$ sample points for each BPC problem. Thus if we try to solve for $\vec{x_v}$ before $\vec{x_u}$, $0 \leq u < v$, are solved, we have to take care of $(2n)^v$ possible situations. Apparently more than a polynomial number of processors are needed if v is not a constant. So how about using Han and Igarashi's random-variable tree [H], [HI]? There are now n random variables and 2^n sample points for each BPC problem. The situation seems to be even more difficult to deal with. However, by close examination, we find out that instead we could reduce the number of processors by using Han and Igarashi's random-variable tree.

We now present a scheme which allows several $\vec{x_u}$'s to be fixed in one pass using Han and Igarashi's random-variable tree for each BPC problem.

First we give a sketch of our approach. The incompleteness of the description in this paragraph will be elaborated on in the rest of this section. Let P be the GPC problem we are to solve. P can be decomposed into q BPC problems to be solved sequentially. Let P_u be the uth BPC problem. Imagine that we are to solve P_u, $0 \leq u < k$, in one pass, i.e., we are to fix $\vec{x_0}, \vec{x_1}, \ldots, \vec{x_{k-1}}$ in one pass, with the help of enough processors. For the moment, we can have a random-variable tree T_u and a derandomization tree D_u for P_u, $0 \leq u < k$. In Step j, our algorithm will work on fixing the bits at level $j - u$ in T_u, $0 \leq u \leq \min\{k - 1, j\}$. The computation in each tree D_u proceeds as we have described in the last section. Note that BPC functions $f_{i_v,j_v}(x_{i_v}, x_{j_v})$ depend on the setting of bits x_{i_u}, x_{j_u}, $0 \leq u < v$. The main difficulty with our scheme is that when we are working on fixing $\vec{x_v}$, the $\vec{x_u}$, $0 \leq u < v$, have not been fixed yet. The only information we can use when we are fixing the random variables at level l of T_u is that random variables at levels 0 to $l + c - 1$ are fixed in T_{u-c}, $0 \leq c \leq u$. This information can be accumulated in the pipeline of our algorithm and transmitted on the *bit-pipeline trees*. Fortunately, this information is sufficient for us to speed up the derandomization process without resorting to too many processors. For the sake of a clear exposition, we first describe a CREW derandomization algorithm. We then show how to convert the CREW algorithm to an EREW algorithm.

Suppose we have $c\sum_{i=0}^{k}(m * 4^i)$ processors available, where c is a constant. Assign $cm * 4^u$ processors to work on P_u for $\vec{x_u}$. We shall work on $\vec{x_u}$, $0 \leq u \leq k$, simultaneously in a pipeline. The random-variable tree for P_u (except that for P_0) is not constructed before the derandomization process begins; rather, it is constructed from a forest as the derandomization process proceeds. A forest containing 2^u random-variable trees corresponds to each variable x_{i_u} in P_u because there are 2^u bit patterns for x_{i_j}, $0 \leq j < u$. We use F_u to denote the random-variable forest for P_u. We fix the random bits on the lth level of F_v (for $\vec{x_v}$) under the condition that random bits from level 0 to level $l+c-1$, $0 \leq c \leq v$, in F_{v-c} have already been fixed. We perform this fixing in constant time. The 2^u random-variable trees corresponding to each random variable x_{i_u} are built bottom-up as the derandomization process proceeds. Immediately before the step in which we fix the random bits on the lth level of F_u, the 2^u random-variable trees corresponding to x_{i_u} are constructed up to the lth level. The details of the algorithm for constructing the random-variable trees will be given later in this section.

Consider a GPC function $f_{i,j}(x_i, x_j)$ under the condition stated in the last paragraph. When we start working on $\vec{x_v}$, we should have the BPC functions $f_{i_v,j_v}(x_{i_v}, x_{j_v})$ evaluated and the function values stored in a table. However, because $\vec{x_u}$, $0 \leq u < v$, have not been fixed yet, we have to try out all possible cases. There are a total of 4^v patterns for bits x_{i_u}, x_{j_u}, $0 \leq u < v$. We use 4^v BPC functions for each pair (i, j). We use

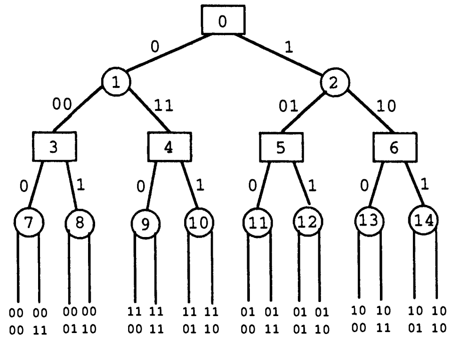


FIG. 4.

$f_{i_v,j_v}(x_{i_v}, x_{j_v})(y_{v-1}y_{v-2}\cdots y_0, z_{v-1}z_{v-2}\cdots z_0)$ to denote the function $f_{i_v,j_v}(x_{i_v}, x_{j_v})$, obtained under the condition that $(x_{i_{v-1}}x_{i_{v-2}}\cdots x_{i_0}, x_{j_{v-1}}x_{j_{v-2}}\cdots x_{j_0})$ is set to $(y_{v-1}y_{v-2}\cdots y_0, z_{v-1}z_{v-2}\cdots z_0)$.

For each pair $(w, w\#0)$ at each level l (this is the level in the random-variable forest), $0 \leq l \leq \log n$, a *bit-pipeline tree* is built (Fig. 4) which is a complete binary tree of height $2k$. Nodes at even depth from the root in a bit-pipeline tree are selectors, and nodes at odd depth are fanout gates. A signal *true* is initially input into the root of the tree and propagates downward toward the leaves. The selectors at depth $2d$ select the output by the decision of the random bits which are the parents of random variables x_{w_d}, $x_{w\#0_d}$ in F_d. One random variable corresponds to each selector. Let random variable r correspond to the selector s. If r is set to 0 then s selects the left child and propagates the true signal to its left child, while no signal is sent to its right child. If r is set to 1 then the true signal will be sent to the right child and no signal will be sent to the left child. If s does not receive any signal from its parent then no signal will be propagated to s's children no matter how r is set. The gates at odd depth in the bit pipeline tree are fanout gates, and pointers from them to their children are labeled with bits which are conditionally set. Refer to Fig. 4, which shows a bit-pipeline tree of height 4. If the selector at the root (node 0) selects 0 (which means that the random variable which is the parent of x_{w_0} and $x_{w\#0_0}$ in the random-variable forest is set to 0), then $x_{w_0} = x_{w\#0_0}$, and therefore the two random variables can only assume the patterns 00 or 11 which are labeled on the pointers from node 1. If, on the other hand, node 0 selects 1 then $x_{w_0} = 1 - x_{w\#0_0}$, the two random variables can only assume the patterns 01 or 10 which are labeled on the pointers of node 2. Let us take node 4 as another example. If node 4 selects 0 then $x_{w_1} = x_{w\#0_1}$; thus the pointers of node 9 are labeled with $\begin{smallmatrix}1&1\\0&0\end{smallmatrix}$ and $\begin{smallmatrix}1&1\\1&1\end{smallmatrix}$. This indicates that the bits for $(w_1w_0, w\#0_1w\#0_0)$ can have two patterns, $(01, 01)$ or $(11, 11)$.

The bit-pipeline tree built for level $\log n$ has height k. No fanout gates will be used. This is a special and simpler case compared to the bit-pipeline trees for other levels. In the following discussion we only consider bit pipeline tree for levels other than $\log n$.

LEMMA 1. *In a bit-pipeline tree there are exactly 2^d nodes at depth $2d$ which will receive the true signal from the root.*

Proof. Each selector selects only one path. Each fanout gate sends the true signal to both children. Therefore, exactly 2^d nodes at depth $2d$ will receive the true signal from the root. □

For each node i at even depth, we shall also say that it has the *conditional-bit pattern* (or *conditional bits*, *bit pattern*), which is the pattern labeled on the pointer from $p(i)$. The root of the bit-pipeline tree has an empty string as its bit pattern.

Define Step 0 as the step when the true signal is input to node 0. The function of a bit-pipeline tree can be described as follows.

Step t: Selectors at depth $2t$ which have received true signals select 0 or 1 for $(w_t, w\#0_t)$. Pass the true signal and the bit-setting information to nodes at depth $2t+2$.

Now consider the selectors at depth $2d$. By Lemma 1, a set of 2^d selectors at depth $2d$ receive the true signal. We call this set the *surviving set* $S^l_{w,d}$. We also denote by $S^l_{w,d}$ the set of bit patterns the 2^d surviving selectors have, where w in the subscript is for $(w, w\#0)$ and l is the level for which the pipeline tree is built. Let selector $s \in S^l_{w,d}$ have bit pattern $(y_{d-1}y_{d-2}\cdots y_0, z_{d-1}z_{d-2}\cdots z_0)$. s compares

$$\begin{aligned} & f^{(l)}_{w_d,w\#0_d}(0,0)(y_{d-1}y_{d-2}\cdots y_0, z_{d-1}z_{d-2}\cdots z_0) \\ & \quad + f^{(l)}_{w_d,w\#0_d}(1,1)(y_{d-1}y_{d-2}\cdots y_0, z_{d-1}z_{d-2}\cdots z_0) \\ & \quad + f^{(l)}_{w\#0_d,w_d}(0,0)(z_{d-1}z_{d-2}\cdots z_0, y_{d-1}y_{d-2}\cdots y_0) \\ & \quad + f^{(l)}_{w\#0_d,w_d}(1,1)(z_{d-1}z_{d-2}\cdots z_0, y_{d-1}y_{d-2}\cdots y_0) \end{aligned}$$

with

$$\begin{aligned} & f^{(l)}_{w_d,w\#0_d}(0,1)(y_{d-1}y_{d-2}\cdots y_0, z_{d-1}z_{d-2}\cdots z_0) \\ & \quad + f^{(l)}_{w_d,w\#0_d}(1,0)(y_{d-1}y_{d-2}\cdots y_0, z_{d-1}z_{d-2}\cdots z_0) \\ & \quad + f^{(l)}_{w\#0_d,w_d}(0,1)(z_{d-1}z_{d-2}\cdots z_0, y_{d-1}y_{d-2}\cdots y_0) \\ & \quad + f^{(l)}_{w\#0_d,w_d}(1,0)(z_{d-1}z_{d-2}\cdots z_0, y_{d-1}y_{d-2}\cdots y_0) \end{aligned}$$

and selects 0 if the former is no less than the latter and 1 otherwise. Note that the selectors which do not receive the true signal (there are $4^d - 2^d$ of them) have bit patterns which are eliminated.

Let $LS^l_{w,d} = \{\alpha | (\alpha, \beta) \in S^l_{w,d}\}$ and $RS^l_{w,d} = \{\beta | (\alpha, \beta) \in S^l_{w,d}\}$.

LEMMA 2. $LS^l_{w,d} = RS^l_{w,d} = \{0, 1\}^d$.

Proof. Lemma 2 is proved by induction. Assume that the lemma is true for bit-pipeline trees of height $2d-2$. A bit-pipeline tree of height $2d$ can be constructed by using a new selector as the root, two new fanout gates at depth 1, and four copies of the bit-pipeline tree of height $2d-2$ at depth 2. If the root selects 0, then patterns 00 and 11 are concatenated with patterns in $S^l_{w,d-1}$; therefore, both $LS^l_{w,d-1}$ and $RS^l_{w,d-1}$ are concatenated with $\{0, 1\}$. The situation when the root selects 1 is similar. □

Now let us consider how functions $f^{(l)}_{i_d,j_d}(x_{i_d}, x_{j_d})(\alpha, \beta)$ are combined. Take the difficult case where both i and j are odd. By Lemma 2, there is only one pattern $p_1 = (\alpha', \alpha) \in S^l_{i\#0,d}$ and there is only one pattern $p_2 = (\beta', \beta) \in S^l_{j\#0,d}$. If the selector having bit pattern p_1 selects 0, then $x_{i_d} = x_{i\#0_d}$ else $x_{i_d} = 1 - x_{i\#0_d}$. If the selector having bit pattern p_2 selects 0, then $x_{j_d} = x_{j\#0_d}$ else $x_{j_d} = 1 - x_{j\#0_d}$. In any case, the conditional-bit pattern is changed to (α', β'), i.e., $f^{(l)}_{i_d,j_d}(x_{i_d}, x_{j_d})(\alpha, \beta)$ will be combined into $f^{(l+1)}_{\lfloor i/2\rfloor_d,\lfloor j/2\rfloor_d}(x_{\lfloor i/2\rfloor_d}, x_{\lfloor j/2\rfloor_d})(\alpha', \beta')$. Note that $x_{\lfloor i/2\rfloor_d}$ and $x_{\lfloor j/2\rfloor_d}$ are new random variables, and here we are not using a superscript to denote this fact. The following lemma ensures that at most four functions will be combined into $f^{(l+1)}_{\lfloor i/2\rfloor_d,\lfloor j/2\rfloor_d}(x_{\lfloor i/2\rfloor_d}, x_{\lfloor j/2\rfloor_d})(\alpha', \beta')$.

Let $S = \{(\alpha', \beta') | (\alpha', \alpha) \in S^l_{i,d}, (\beta', \beta) \in S^l_{j,d}, \alpha, \beta \in \{0, 1\}^d\}$.

LEMMA 3. $|S| = 4^d$.

Proof. The definition of S can be viewed as a linear transformation. Represent $x \in \{0, 1\}^d$ by a vector of 2^d bits with the xth bit set to 1 and the rest of the bits set to 0. The transformation $\alpha \mapsto \alpha'$ can be represented by a permutation matrix of order 2^d. The transformation $(\alpha, \beta) \mapsto (\alpha', \beta')$ can be represented by a permutation matrix of order 2^{d+1}. □

Lemma 3 tells us that the functions to be combined are permuted; therefore, no more than four functions will be combined under any conditional-bit pattern.

We call this scheme of combining *combining functions with respect to the surviving set.*

We have completed a preliminary description of our derandomization scheme for the GPC problem. The algorithm for processors working on $\vec{x}_d$, $0 \leq d < k$, can be summarized as follows.

Step t $(0 \leq t < d)$: Wait for the pipeline to be filled.

Step $d + t$ $(0 \leq t < \log n)$: Fix random variables at level t for all conditional-bit patterns in the surviving set. (*There are 2^d such patterns in the surviving set.*) Combine functions with respect to the surviving set. (*At the same time the bit-setting information is transmitted to the nodes at depth $2d + 2$ on the bit-pipeline tree.*)

Step $d + \log n$: Fix the only remaining random variable at level $\log n$ for the only bit pattern in the surviving set. Output the good point for $\vec{x}_d$. (*At the same time, the bit-setting information is transmitted to the node at depth $d + 1$ on the bit-pipeline tree.*)

THEOREM 1. *The GPC problem can be solved on the CREW PRAM in time $O((q/k + 1)(\log n + k + \tau))$ with $O(4^k m)$ processors, where τ is the time for a single processor to evaluate a BPC function $f_{i_d, j_d}(x_{i_d}, x_{j_d})(\alpha, \beta)$.*

Proof. The correctness of the scheme comes from the fact that as random bits are fixed, a smaller space with higher expectation is obtained, and thus when all random bits are fixed, a good point is found. To solve the uth BPC problem is to evaluate $P_u(\vec{z}) = E[B(\vec{x}_{q-1} \cdots \vec{x}_{u+1} \vec{z} \vec{y})]$, $\vec{z} \in \{0, 1\}^n$, where $\vec{y} = < y_i \in \{0, 1\}^u : i = 0, 1, \ldots, n-1 >$ is fixed. We then view $\vec{z}$ as a random variable uniformly distributed on $\{0, 1\}^n$ and find $\vec{z}'$ such that $P_u(\vec{z}') \geq E[P_u(\vec{z})]$. If we have a huge number of processors, we could solve all BPC problems in parallel by solving each P_u with all possible $\vec{y}$'s. Such an algorithm is apparently correct. In our scheme, $P_u(\vec{z})$ is evaluated by evaluating $E[f_{i,j}(\alpha\beta, \alpha'\beta')]$, $\alpha, \alpha' \in \{0, 1\}$, $\beta, \beta' \in \{0, 1\}^u$. This is guaranteed to be correct by linearity of expectation. We use a pipeline to solve the P_u's. Thus our algorithm is still correct while the number of processors needed is drastically reduced.

With $O(4^k m)$ processors, k $\vec{x}_u$'s are fixed in one pass. Each pass takes $O(\log n + k + \tau)$ time, τ for evaluating BPC functions (i.e., setting up the function tables for the BPC problems) and $O(\log n + k)$ time for fixing all random bits on the random-variable trees. The time complexity for solving the GPC problem is $O((q/k + 1)(\log n + k + \tau))$. □

We have not yet discussed explicitly the way the random-variable trees are constructed. The construction is implied in the surviving set we computed. We now give the algorithm for constructing the random variable trees. This algorithm will help better understand the whole scheme.

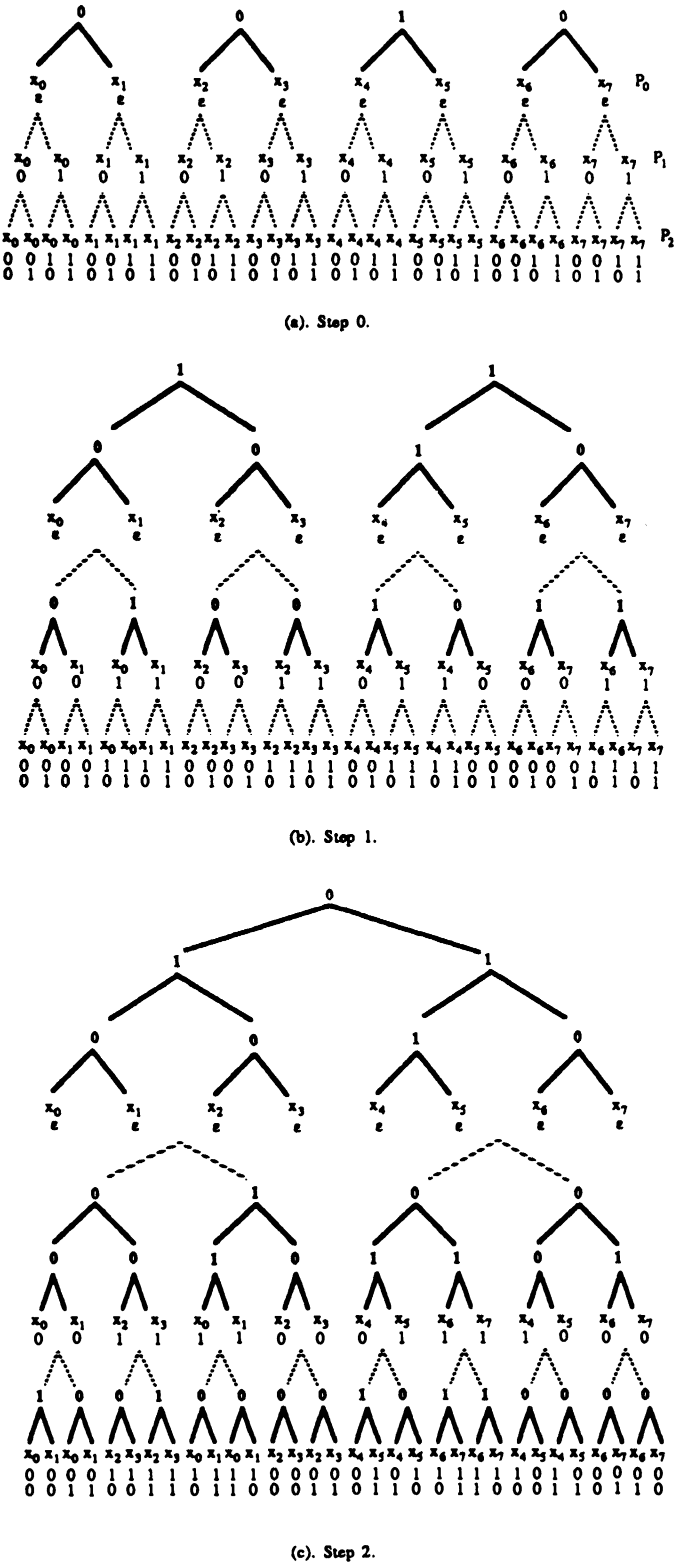
The ith node under conditional-bit pattern j at the lth level of the random-variable trees for P_u is stored in $T_u^{(l)}[i][j]$. The leaves are stored in $T_u^{(-1)}$. Initially, bit-pipeline trees for level -1 are built such that $T_u^{(-1)}[i][j]$ has two children $T_{u+1}^{(-1)}[i][j0]$, $T_{u+1}^{(-1)}[i][j1]$, where $j0$ and $j1$ are the concatenations of j with 0 and 1, respectively. Note that the bit-pipeline tree constructed here is different from the one we built before, but in principle they are the same tree and perform the same function in our scheme. The algorithm for constructing the random-variable trees for P_u is below.

```
PROCEDURE RV-TREE
  begin
    Step $t$ $(0 \leq t < u)$: Wait for the pipeline to be filled.

    Step $u + t$ $(0 \leq t < \log n)$:
      (*In this step, we will build $T_u^{(t)}[i][j]$, $0 \leq i < n/2^{t+1}$, $0 \leq j < 2^u$. At the
      beginning of this step, $T_{u-1}^{(t)}[i][j]$ has already been constructed. Let $T_{u-1}^{(t-1)}[i0][j]$
      and $T_{u-1}^{(t-1)}[i1][j']$ be the two children of $T_{u-1}^{(t)}[i][j]$ in the random-variable tree.
      $T_u^{(t-1)}[i0][j0]$ and $T_u^{(t-1)}[i0][j1]$ are the children of $T_{u-1}^{(t-1)}[i0][j]$, and $T_u^{(t-1)}[i1][j'0]$
      and $T_u^{(t-1)}[i1][j'1]$ are the children of $T_{u-1}^{(t-1)}[i1][j']$ in the bit-pipeline tree for level
      $t-1$. The setting of the random variable $r$ for the pair $(i0, i1)$ at level $t$ for $P_{u-1}$,
      i.e., the random variable in $T_{u-1}^{(t)}[i][j]$, is known.*)

      make $T_u^{(t-1)}[i0][j0]$ and $T_u^{(t-1)}[i1][j'r]$ the children of $T_u^{(t)}[i][j0]$ in the random-
      variable forest for $P_u$; (*$jr$ is the concatenation of $j$ and $r$.*)

      make $T_u^{(t-1)}[i0][j1]$ and $T_u^{(t-1)}[i1][j'\bar{r}]$ the children of $T_u^{(t)}[i][j1]$ in the random-
      variable forest for $P_u$; (*$\bar{r}$ is the complement of $r$.*)

      make $T_u^{(t)}[i][j0]$ and $T_u^{(t)}[i][j1]$ the children of $T_{u-1}^{(t)}[i][j]$ in the bit-pipeline tree
      for level $t$;

      fix the random variables in $T_u^{(t)}[i][j0]$ and $T_u^{(t)}[i][j1]$;

    Step $u + \log n$:
      (*At the beginning of this step, the random-variable trees have been built for $T_i$,
      $0 \leq i < u$. Let $T_{u-1}^{(\log n)}[0][j]$ be the root of $T_{u-1}$. The random variable $r$ in
      $T_{u-1}^{(\log n)}[0][j]$ has been fixed. In this step, we will choose one of the two children of
      $T_{u-1}^{(\log n)}[0][j]$ in the bit-pipeline tree for level $\log n$ as the root of $T_u$.*)

      make $T_u^{(\log n-1)}[0][jr]$ the child of $T_u^{(\log n)}[0][jr]$ in the random-variable tree;

      make $T_u^{(\log n)}[0][jr]$ the child of $T_{u-1}^{(\log n)}[0][j]$ in the bit-pipeline tree for level $\log n$;

      fix the random variable in $T_u^{(\log n)}[0][jr]$;

      output $T_u^{(\log n)}[0][jr]$ as the root of $T_u$;
  end
```

Procedure RV-Tree uses the pipelining technique as well as a dynamic-programming technique. These are some of the essential elements of our scheme.

An example of the execution of Procedure RV-Tree is shown in Fig. 5.

We now show how to remove the concurrent-read feature from the scheme. The difficulty here is in the step of combining functions with respect to the surviving set. The size of the surviving set $S_{u,k}^l$ is 2^k while there are 4^k conditional-bit patterns. There are 4^k functions $f_{u_k,v_k}^{(l)}(x_{u_k}, x_{v_k})$, one for each bit pattern (α, β). All 4^k functions will consult the surviving set

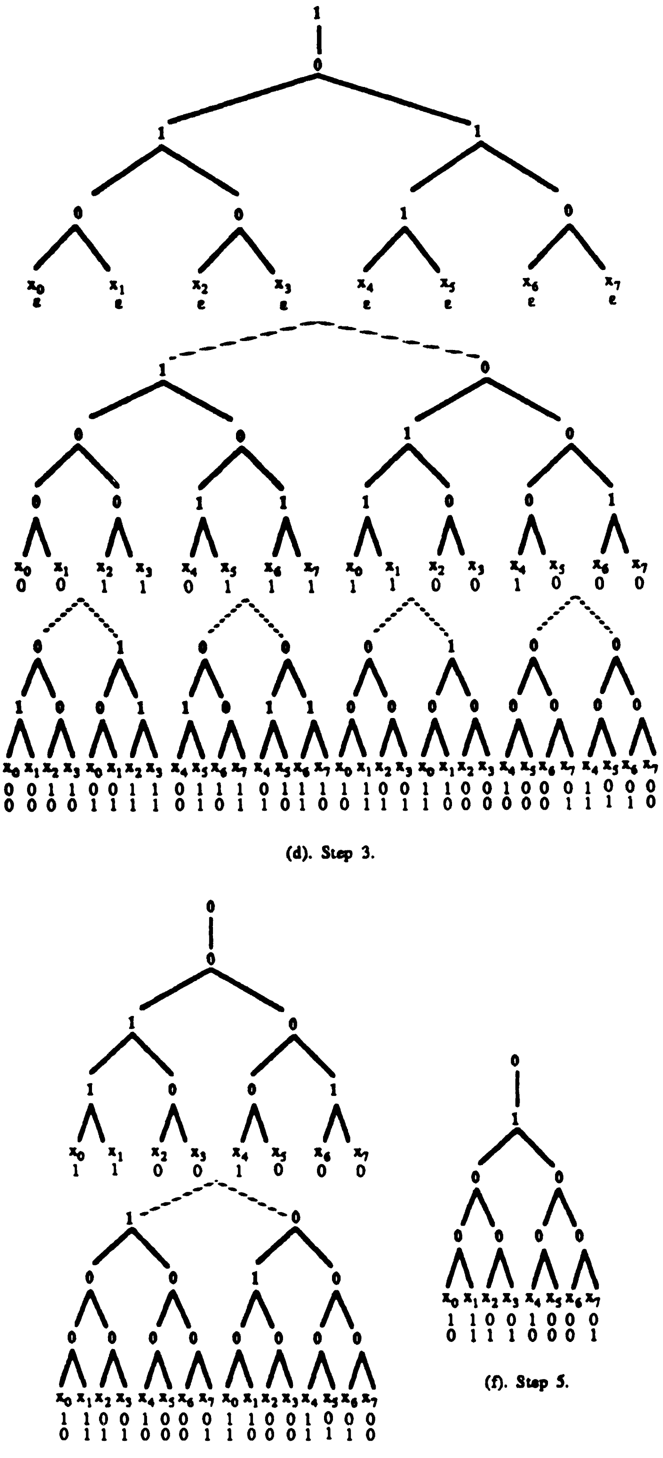


(a). Step 0.

(b). Step 1.

(c). Step 2.

FIG. 5. *An execution of RV-Tree. Darkened lines and bits in boldface are random-variable trees. Dotted lines are bit-pipeline trees.*



(d). Step 3.

(e). Step 4.

(f). Step 5.

FIG. 5. (*Cont.*)

in order for them to be combined into new functions. The problem is how to do it in constant time without resorting to concurrent read.

We show how to let $f^{(l)}_{u_k,u\#0_k}(x_{u_k}, x_{u\#0_k})(\alpha, \beta)$ to acquire the bit pattern α' which satisfies $(\alpha', \beta) \in S^l_{u,k}$. Function $f^{(l)}_{u_k,v_k}(x_{u_k}, x_{v_k})(\alpha, \beta)$ can then obtain the bit pattern α' from $f^{(l)}_{u_k,u\#0_k}(x_{u_k}, x_{u\#0_k})(\alpha, \beta)$ by the pipeline scheme described in [H].

Suppose we are to solve P_u, $0 \leq u \leq k$, in one pass. We solve 4^k copies of P_{k-1}; one copy corresponds to one conditional-bit pattern in P_k. $f^{(l)}_{u_k,v_k}(x_{u_k}, x_{v_k})(\alpha, \beta)$ in P_k can obtain α' by following the computation in the copy of P_{k-1} which corresponds to (α, β). This can be done without concurrent read. Now for each of the 4^k copies of P_{k-1}, we solved 4^{k-1} copies of P_{k-2}; one copy corresponds to one conditional-bit pattern in P_{k-1}, and so on. Thus to remove concurrent read we need $c^{k^2}(m+n)$ processors for solving P_u, $0 \leq u \leq k$, in one pass, where c is a suitable constant. Note also that it takes $O(k^2)$ time to make needed copies.

THEOREM 2. *The GPC problem can be solved on the EREW PRAM in time $O((q/\sqrt{k} + 1)(\log n + k + \tau))$ with $O(c^k m)$ processors, where c is a suitable constant and τ is the time for a single processor to evaluate a BPC function $f_{u_d,v_d}(x_{u_d}, x_{v_d})(\alpha, \beta)$.*

4. Δ + 1 vertex coloring.

We apply our scheme to Luby's formulation of the $\Delta + 1$ vertex-coloring problem [L2], [L3]. First we adapt his formulation and then apply our fast derandomization scheme to obtain a faster algorithm. Luby showed [L2], [L3] that after solving a GPC problem a constant fraction of the vertices can be deleted. The main change now is to show that after solving a GPC problem a constant fraction of the edges can be deleted. We follow the notations and definitions as given by Luby [L2], [L3].

Let $G = (V, E)$ be the graph we are to color. Let $adj(i)$ be the set of vertices which are adjacent to vertex i, and let $d(i)$ be the degree of vertex i. Let $avail_i$ be the set of colors which can be used to color vertex i, and let $Navail_i = |avail_i|$. Let k_i be such that $2^{k_i-1} < 4Navail_i \leq 2^{k_i}$ and let $Nlist_i = 2^{k_i}$. Let $list_i[0, \ldots, Nlist_i - 1]$ be an array such that the first $Navail_i$ entries in $list_i$ are the elements of $avail_i$ in sorted order and the remaining entries in $list_i$ have value Λ. Let $q = \max\{k_i | i \in V\}$. Let $\vec{x} = < x_i \in \{0, 1\}^q, i \in V >$. For $i \in V$, let $list_i(x_i)$ be the entry in $list_i$ indexed by the first k_i bits of x_i. Also define the following functions.

For all $i \in V$, let

$$Y_i(x_i) = \begin{cases} 1 \text{ if } list_i(x_i) \in avail_i, \\ 0 \text{ if } list_i(x_i) = \Lambda. \end{cases}$$

For all $(i, j) \in E$, let

$$Y_{i,j}(x_i, x_j) = \begin{cases} -1 \text{ if } list_i(x_i) = list_j(x_j) \neq \Lambda, \\ 0 \text{ otherwise.} \end{cases}$$

The BENEFIT function B is defined as

$$B(\vec{x}) = \sum_{i \in V} \frac{d(i)}{2} \left(Y_i(x_i) + \sum_{j \in adj(i)} Y_{i,j}(x_i, x_j) \right).$$

Function B sets a lower bound on the number of edges deleted [L2], [L3] should the vertex i be tentatively assigned color $list_i(x_i)$. We will not repeat the definitions of the auxiliary functions TY_i and $TY_{i,j}(x_i, x_j)$, since their definitions can be found in [L2] and [L3]. The auxiliary function TB is now defined as

$$TB\,(\vec{x}) = \sum_{i \in V} \frac{d(i)}{2} \left(TY_i\,(x_i) + \sum_{j \in adj(i)} TY_{i,j}\,(x_i, x_j) \right).$$

Following Luby's proof [L2], [L3], we have $TY_i(\Lambda) \geq 1/8$, $TY_{i,j}(\Lambda, \Lambda) \geq -1/16$, and therefore we have the following lemma.

LEMMA 4. $TB\ (\overrightarrow{\Lambda}) \geq |E|/16$.

Thus by solving a GPC problem,[1] we are guaranteed to eliminate a constant fraction of the edges.

Let $c^k m$ be the number of processors needed to compute k BPC problems in a GPC problem in one pass. There will be $O(\log n)$ stages in the modified algorithm. Each stage contains a constant number of GPC problems and reduces the number of edges so that there will be no more than a $1/c$ fraction of the edges left. Therefore, during stage i there will be e edges in the remaining graph and $c^i e$ processors available. Because each stage has $O(\log n)$ BPC problems, the time complexity for stage i is $O(\log^2 n/i)$. Thus the time complexity of the whole algorithm becomes $O(\sum_{i=1}^{O(\log n)} \log^2 n/i) = O(\log^2 n \log\log n)$.

The number of processors used in the algorithm can be reduced to $O((m+n)/\log\log n)$. We examine the first $O(\log\log\log n)$ stages. In stage i, we can have $c^i/\log\log n$ processors for each edge under each conditional-bit pattern. Therefore, the tables for the BPC functions in stage i can be computed in time $O(\log^2 n \log\log n/c^i)$, and the overall time for table construction for the whole algorithm is $O(\log^2 n \log\log n)$. The calculation for the time for constructing the derandomization trees is similar and can be shown to be $O(\log^2 n \log\log n)$ with $O((m+n)/\log\log n)$ processors. In the first $O(\log\log\log n)$ stages, our GPC algorithm will be invoked with $k = 1$. The time complexity for these stages is

$$O\left(\sum_{i=0}^{O(\log\log\log n)} \frac{\log^2 n \log\log n}{c^i}\right) = O(\log^2 n \log\log n).$$

The remaining stages take $O(\log^2 n \log\log n)$ time by the analysis in the last paragraph.

THEOREM 3. *There is a CREW PRAM algorithm for the* $\Delta + 1$ *vertex-coloring problem with time complexity* $O(\log^2 n \log\log n)$ *using* $O((m+n)/\log\log n)$ *processors.*

We also have, the following theorem.

THEOREM 4. $O(mn^\epsilon)$ *processors are sufficient to solve the* $\Delta + 1$ *vertex-coloring problem in time* $O(\log^2 n)$ *on the CREW PRAM, where* $\epsilon > 0$ *is an arbitrary constant.*

Proof. This is because one GPC problem can now be solved in $O(\log n)$ time. □

5. Maximal independent set. Let $G = (V, E)$ be an undirected graph. For $W \subseteq V$, let $N(W) = \{i \in V \mid \exists\ j \in W, (i, j) \in E\}$. Known parallel algorithms [ABI], [KW], [GS1], [GS2], [L1], [L3] for computing a maximal independent set have the following form.

```
PROCEDURE GENERAL-INDEPENDENT
begin
      $I := \phi$;
      $V' := V$;
      while $V' \neq \phi$ do
         begin
            Find an independent set $I' \subseteq V'$;
            $I := I \cup I'$;
            $V' := V' - (I' \cup N(I'))$;
         end
end
```

[1] The problem formulated [L2], [L3] resembles a GPC problem. It is not a GPC problem in the strict sense. For our purpose, we may view it as a GPC problem because our GPC algorithm applies.

Luby's work [L3] formulated each iteration of the while loop in General-Independent as a GPC problem. We now adapt Luby's formulation [L1], [L3].

Let k_i be such that $2^{k_i-1} < 4d(i) \leq 2^{k_i}$. Let $q = \max\{k_i | i \in V\}$. Let $\vec{x} =< x_i \in \{0,1\}^q, i \in V\}$. The length $|x_i|$ of x_i is defined to be k_i. Define[2]

$$Y_i(x_i) \quad = \begin{cases} 1 & \text{if } x_i(|x_i|-1)\cdots x_i(0) = 0^{|x_i|}, \\ 0 & \text{otherwise,} \end{cases}$$

$$Y_{i,j}(x_i, x_j) = -Y_i(x_i)Y_j(x_j),$$

$$B(\vec{x}) \quad = \sum_{i \in V} \frac{d(i)}{2}$$

$$\sum_{j \in adj(i)} \left(Y_j(x_j) + \sum_{k \in adj(j), d(k) \geq d(j)} Y_{j,k}(x_j, x_k) + \sum_{k \in adj(i)-\{j\}} Y_{j,k}(x_j, x_k) \right),$$

where $x_i(p)$ is the pth bit of x_i.

Function B sets a lower bound on the number of edges deleted from the graph [L1], [L3] should vertex i be tentatively labeled as an independent vertex if $x_i = (0 \cup 1)^{q-|x_i|}0^{|x_i|}$. The following lemma was proven in [L1, Thm. 1].

LEMMA 5 [L1]. *$E[B] \geq |E|/c$ for a constant $c > 0$.*

Function B can be written as

$$B(\vec{x}) = \sum_{j \in V} \left(\sum_{i \in adj(j)} \frac{d(i)}{2} \right) Y(x_j) + \sum_{(j,k) \in E, d(k) \geq d(j)} \left(\sum_{i \in adj(j)} \frac{d(i)}{2} \right) Y_{j,k}(x_j, x_k)$$

$$+ \sum_{i \in V} \frac{d(i)}{2} \sum_{j,k \in adj(i), j \neq k} Y_{j,k}(x_j, x_k)$$

$$= \sum_i f_i(x_i) + \sum_{(i,j)} f_{i,j}(x_i, x_j),$$

where

$$f_i(x_i) = \left(\sum_{j \in adj(i)} \frac{d(j)}{2} \right) Y(x_i)$$

and

$$f_{i,j}(x_i, x_j) = \delta(i,j) \left(\sum_{k \in adj(i)} \frac{d(k)}{2} \right) Y_{i,j}(x_i, x_j) + \left(\sum_{k \in V \text{ and } i,j \in adj(k)} \frac{d(k)}{2} \right) Y_{i,j}(x_i, x_j),$$

$$\delta(i,j) \quad = \begin{cases} 1 & \text{if } (i,j) \in E \text{ and } d(j) \geq d(i), \\ 0 & \text{otherwise.} \end{cases}$$

Thus each execution of a GPC procedure eliminates a constant fraction of the edges from the graph. It takes $O(M(n))$ (which is currently $O(n^{2.376})$ [CW]) processors to compute a

[2] In Luby's formulation [L3], $Y_i(x_i)$ is zero unless the first $|x_i|$ bits of x_i are 1's. In order to be consistent with the notations in our algorithm, we let $Y_i(x_i)$ be zero unless the first $|x_i|$ bits of x_i are 0's.

matrix multiplication in time $O(\log n)$ to arrive at the GPC functions f_i's and $f_{i,j}$'s because of the term $\sum_{i \in V} \frac{d(i)}{2} \sum_{j,k \in adj(i), j \neq k} Y_{j,k}(x_j, x_k)$ in function B. We organize our algorithm for the maximal independent set problem into $O(\log n)$ stages such that in stage i, the graph has no more than $|E|/c^i$ vertices and a constant number of GPC problems will be solved in stage i. By Theorem 1, we achieve time complexity $O(\log^2 n)$.

THEOREM 5. *There is a CREW PRAM algorithm for the maximal independent set problem with time complexity $O(\log^2 n)$ using $O(M(n))$ processors.*

Proof. The time and processor complexities for computing matrix multiplication dominate. □

We will give a second algorithm for the maximal independent set problem. We take advantage of the special properties of the GPC functions to reduce the number of processors to $O(m + n)$. We cannot use the derandomization scheme in §3 directly because it would involve a matrix multiplication, as we have seen in the design of our first maximal independent set algorithm. The structure of our second algorithm is complicated. We first give an overview of the algorithm.

5.1. Overview of the second algorithm. Because we can reduce the number of edges by a constant fraction after solving a GPC problem, a maximal independent set will be computed after $O(\log n)$ GPC problems are solved. Our algorithm has two stages, the initial stage and the speedup stage. The initial stage consists of the first $O(\log^{0.5} n)$ GPC problems. Each GPC problem is solved in $O(\log^2 n)$ time. The time complexity for the initial stage is thus $O(\log^{2.5} n)$. When the first stage finishes, the remaining graph has size $O((m + n)/2^{\sqrt{\log n}})$. There are $O(\log n)$ GPC problems in the speedup stage. A GPC problem of size s in the speedup stage is solved in time $O(\log^2 n/\sqrt{k})$ with $O(c^k s \log n)$ processors. Therefore the time complexity of the speedup stage is $O(\sum_{i=O(\sqrt{\log n})}^{O(\log n)} (\log^2 n/\sqrt{i})) = O(\log^{2.5} n)$. The initial stage is mainly to reduce the processor complexity while the speedup stage is mainly to reduce the time complexity.

We used matrix multiplication in our first algorithm because of the term $\sum_{i \in V} \frac{d(i)}{2} \cdot \sum_{j,k \in adj(i), j \neq k} Y_{j,k}(x_j, x_k)$ in function B. We shall call this term the vertex-cluster term. There is a cluster $C(v) = \{x_w | (v, w) \in E\}$ for each vertex v. Alternatively we may use $O(\sum_{v \in V} d^2(v))$ processors, $d^2(v)$ processors for cluster $C(v)$, to evaluate all GPC functions and to apply our derandomization scheme given in §3. However, to reduce the number of processors to $O(m + n)$ we have to use a modified version of our derandomization scheme in §3.

Consider the problem of fixing a random-variable r in the random-variable tree. We did this in constant time in §3 (Theorem 1). We now outline how r is fixed in the initial stage. We cannot do it in constant time because the GPC function $f(x, y)$, where x and y are the leaves in the subtree rooted at r, is in fact the sum of several functions scattered in the second term of function B and in several clusters. We will not combine BPC functions in the derandomization process. As we have explained in §2, setting r requires $O(\log n)$ time because of the summation of function values. (Note that the summation of n items can be done in time $O(n/p + \log n)$ time with p processors.) A BPC problem takes $O(\log^2 n)$ time to solve. We pipeline all BPC problems in a GPC problem and get time complexity $O(\log^2 n)$ for solving a GPC problem.

The functions in B have a special property which we will exploit in our algorithm. Each variable x_i has a length $|x_i| \leq q = O(\log n)$. $Y_{i,j}(x_i, x_j)$ is 0 unless the first $|x_i|$ bits of x_i are 0's and the first $|x_j|$ bits of x_j are 0's. When we apply our scheme, there is no need to keep BPC functions $Y_{i_u,j_u}(x_{i_u}, x_{j_u})$ for all conditional-bit patterns because many of these patterns will yield zero BPC functions. In our algorithm, we keep one copy of $Y_{i_u,j_u}(x_{i_u}, x_{j_u})$ with conditional bits set to 0's. This of course helps reduce the number of processors. In particular, the random-variable tree for each P_u now requires at most $O(n)$ processors, instead of $O(c^u n)$ processors as we have used in §3 on the CREW PRAM.

There are $d^2(i)$ BPC functions in cluster $C(i)$, while we can allocate at most $d(i)$ processors in the very first GPC problem because we have at most $O(m+n)$ processors for the GPC problem. What we do is use an *evaluation tree* for each cluster. The evaluation tree $TC(i)$ for cluster $C(i)$ is a "subtree" of the random-variable tree. The leaves of $TC(i)$ are the variables in $C(i)$. An interior node of the random-variable tree is not present in $TC(i)$ if none of the leaves of the subtree rooted at the interior node is in $C(i)$. When we are fixing r, if r is not in $TC(i)$ then cluster $C(i)$ does not contribute anything. If r is in $TC(i)$ then the contribution of $C(i)$ can be obtained by evaluating the function $f(x, y)$, where x and y are leaves in the evaluation subtree rooted at r and x and y are in different subtrees of r. If r has a leaves in the left subtree and b leaves in the right subtree then the contribution from $TC(i)$ for fixing r is the sum of ab function values. We will give the details of evaluating this sum using a constant number of operations.

Let us summarize the main ideas. We do not combine functions and achieve time $O(\log^2 n)$ for solving a BPC problem. We put all BPC problem in a GPC problem as one batch into a pipeline to get $O(\log^2 n)$ time for solving a GPC problem. We use a special property of functions in B to maintain one copy for each BPC function for only conditional bits of all 0's. We use evaluation trees to take care of the vertex cluster term.

We now sketch the speedup stage. Since we have to solve $O(\log n)$ GPC problems in this stage, we have to reduce the time complexity for a GPC problem to $o(\log^2 n)$ in order to obtain $o(\log^3 n)$ time. We use a modified random-variable tree as shown in Fig. 3(c) in §2. Such a random-variable tree has $S = \lceil(\log n + 1)/a\rceil$ blocks. Each block contains a levels. We fix a block in a step instead of fixing a level in a step. Each step takes $O(\log n)$ time and a BPC problem takes $O(S \log n)$ time. If we have as many processors as we want, we could solve all BPC problems in a GPC problem by enumerating all possible cases instead of putting them through a pipeline; i.e., in solving P_u, we could guess all possible settings of random variables for P_v, $0 \leq v < u$. We have explained this approach in the proof of Theorem 1 in §2. In doing so we would achieve time $O(S \log n)$ for solving a GPC problem. In reality, we have extra processors, but they are not enough for us to enumerate all possible situations. We therefore put a BPC problems of a GPC problem in a team. All BPC problems in a team are solved by enumeration. Thus they are solved in time $O(S \log n)$. Let b be the number of teams we have. We put all these teams into a pipeline and solve them in time $O((S+b)\log n)$. The approach of the speedup stage can be viewed as that of the initial stage with added parallelism which comes with the help of extra processors.

5.2. The initial stage. We first show how to solve a GPC problem for function B in time $O(\log^2 n)$ using $O((m+n)\log n)$ processors.

$O(m+n)$ processors will be allocated to each BPC problem. The algorithm for processors working on F_u has the following form.

Step t $(0 \leq t < u)$: Wait for the pipeline to be filled.

Step $u + t$ $(0 \leq t < \log n)$: Fix random variables at level t.

Step $u + \log n$: Fix the only remaining random variable at level $\log n$. Output the good point for $\vec{x}_u$.

We will allow $O(\log n)$ time for each step and $O(\log^2 n)$ time for the whole algorithm. Note that we do not combine functions with respect to the surviving set and therefore use $O(\log n)$ time for a step.

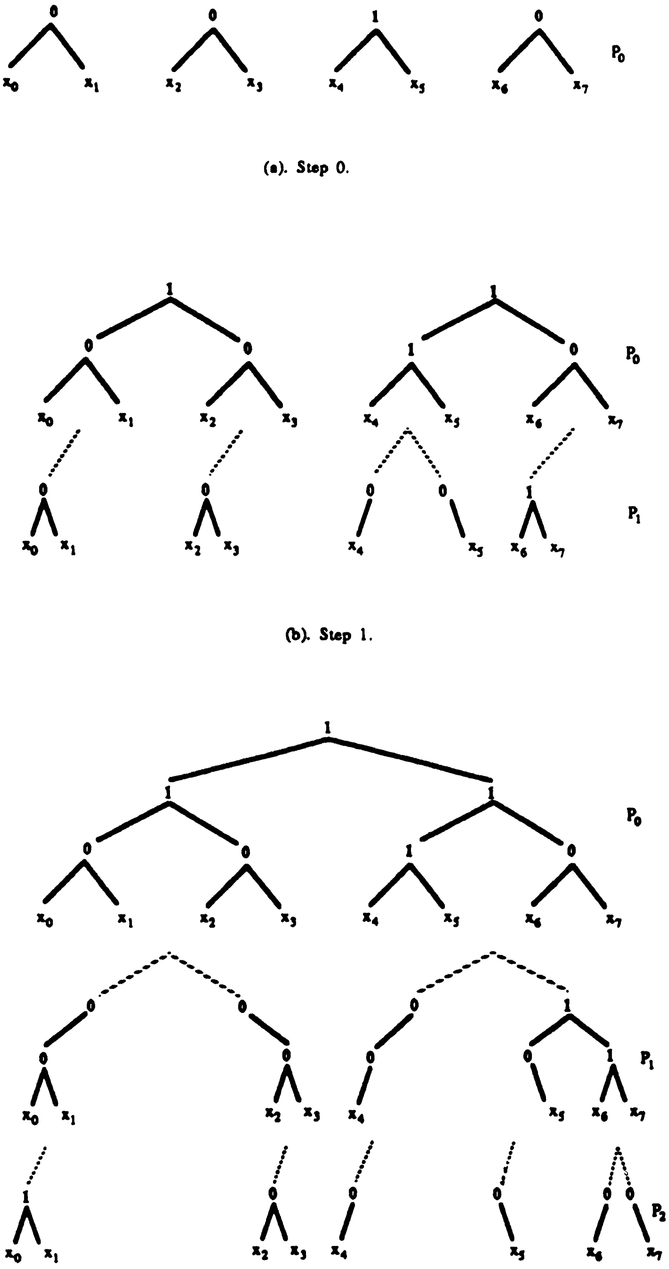
The way T_u is constructed can be described by algorithm MRV-Tree, a modified version of algorithm RV-Tree in §3. In MRV-Tree we do not enumerate all possible conditional-bit patterns. Only the bit pattern of all 0's is kept. Thus a node on a bit-pipeline tree may not have both children. A variable x_i only appears in F_u as x_{i_u} with $u < |x_i|$ because the setting of random variables in F_u, $u \geq |x_i|$, is not affected by x_{i_u}.

```
PROCEDURE MRV-TREE
  begin
    Step $t$ $(0 \leq t < u)$: Wait for the pipeline to be filled.

    Step $u$:
      (*In this step, we will build $T_u^{(0)}[i][j]$. $0 \leq i < n/2$ and $j$ are indices for which
      $T_u^{(0)}[i][j]$ is not empty. At the beginning of this step, $T_{u-1}^{(0)}[i][j]$ has already been
      constructed if it is not empty. Random variables $x_i$ have been transmitted to depth
      $u$ of the bit pipeline tree for level 0.*)

    for each node $T_u^{(0)}[i][j]$
      if $T_u^{(0)}[i][j]$ has received either $x_{2i}$ or $x_{2i+1}$ from $T_{u-1}^{(0)}[i][j/2]$
        (*$x_{2i}$ and $x_{2i+1}$ becomes $x_{2i_u}$ and $x_{2i+1_u}$ in $T_u$.*) then
        begin
          if $T_u^{(0)}[i][j]$ has received $x_{2i}$ then
            make it the left child of $T_u^{(0)}[i][j]$ in the random-variable forest for $P_u$;

          if $T_u^{(0)}[i][j]$ has received $x_{2i+1}$ then
            make it the right child of $T_u^{(0)}[i][j]$ in the random-variable forest for $P_u$;

          fix random variable $r$ in $T_u^{(0)}[i][j]$;

          make $T_u^{(0)}[i][j]$ a child of $T_{u-1}^{(0)}[i][j/2]$ in the bit-pipeline tree;

          if $T_u^{(0)}[i][j]$ has received $x_{2i}$ and $|x_{2i}| \geq u$ then
            transmit $x_{2i}$ to $T_{u+1}^{(0)}[i][j0]$;

          if $T_u^{(0)}[i][j]$ has received $x_{2i+1}$ and $|x_{2i+1}| \geq u$ then
            transmit $x_{i\#0}$ to $T_{u+1}^{(0)}[i][jr]$;
        end
      else (*$T_u^{(0)}[i][j]$ is empty.*);

    Step $u+t$ $(1 \leq t < \log n)$:
      (*In this step, we will build $T_u^{(t)}[i][j]$, $0 \leq i < n/2^{t+1}$ and $j$ are indices for
      which $T_u^{(t)}[i][j]$ is not empty. At the beginning of this step, $T_{u-1}^{(t)}[i][j]$ has already
      been constructed. Let $T_{u-1}^{(t-1)}[i0][j]$ and $T_{u-1}^{(t-1)}[i1][j']$ (they may be empty) be the
      two children in the random-variable subtree rooted at $T_{u-1}^{(t)}[i][j]$. $T_u^{(t-1)}[i0][j0]$
      and $T_u^{(t-1)}[i0][j1]$ (they may be empty) are the children of $T_{u-1}^{(t-1)}[i0][j]$, and
      $T_u^{(t-1)}[i1][j'0]$ and $T_u^{(t-1)}[i1][j'1]$ (they may be empty) are the children of
      $T_{u-1}^{(t-1)}[i1][j']$ in the bit pipeline tree for level $t-1$. The setting of the random
      variable $r$ for the pair $(i0, i1)$ at level $t$ for $P_{u-1}$, i.e., the random variable in
      $T_{u-1}^{(t)}[i][j]$, is known.*)
```

```
if $T_u^{(t-1)}[i0][j0]$ is not empty then
    make it the left child of $T_u^{(t)}[i][j0]$ in the random-variable forest for $P_u$;

if $T_u^{(t-1)}[i1][j'r]$ is not empty then
    make it the right child of $T_u^{(t)}[i][j0]$ in the random-variable forest for $P_u$;

if $T_u^{(t-1)}[i0][j1]$ is not empty then
    make it the left child of $T_u^{(t)}[i][j1]$ in the random-variable forest for $P_u$;

if $T_u^{(t-1)}[i1][j'\bar{r}]$ is not empty then
    make it the right child of $T_u^{(t)}[i][j1]$ in the random-variable forest for $P_u$;

if $T_u^{(t)}[i][j0]$ is not empty then
    make it the left child of $T_{u-1}^{(t)}[i][j]$ in the bit-pipeline tree for level $t$;

if $T_u^{(t)}[i][j1]$ is not empty then
    make it the right child of $T_{u-1}^{(t)}[i][j]$ in the bit-pipeline tree for level $t$;

fix the random variables in $T_u^{(t)}[i][j0]$ and $T_u^{(t)}[i][j1]$;

Step $u + \log n$:
    (*At the beginning of this step, the random-variable trees have been built for $T_i$, $0 \le i < u$. Let $T_{u-1}^{(\log n)}[0][j]$ be the root of $T_{u-1}$. The random variable $r$ in $T_{u-1}^{(\log n)}[0][j]$ has been set. In this step, we will choose one of the two children of $T_{u-1}^{(\log n)}[0][j]$ in the bit-pipeline tree for level $\log n$ as the root of $T_u$.*)

    if $T_u^{(\log n-1)}[0][jr]$ is not empty then
        begin
            make $T_u^{(\log n-1)}[0][jr]$ the child of $T_u^{(\log n)}[0][jr]$ in the random-variable tree;

            make $T_u^{(\log n)}[0][jr]$ the child of $T_{u-1}^{(\log n)}[0][j]$ in the bit-pipeline tree for level $\log n$;

            fix the random variable in $T_u^{(\log n)}[0][jr]$;

            output $T_u^{(\log n)}[0][jr]$ as the root of $T_u$;
        end
end
```

Note that i and j in $T_u^{(t)}[i][j]$ are parameters and $T_u^{(t)}$ is not a two dimensional array here. We can view algorithm MRV-Tree as one which distributes random variables x_i into different sets. Each set is indexed by (u, t, i, j). We call these sets BD sets because they are obtained on the bit-pipeline trees and the derandomization trees. x is in $BD(u, t, i, j)$ if x is a leaf in $T_u^{(t)}[i][j]$. When u and t are fixed, $BD(u, t, i, j)$ sets are disjoint. Because we allow $O(\log n)$ time for each step in MRV-Tree, the time complexity for constructing the random-variable trees is $O(\log^2 n)$.

See Fig. 6 for an execution of MRV-Tree.

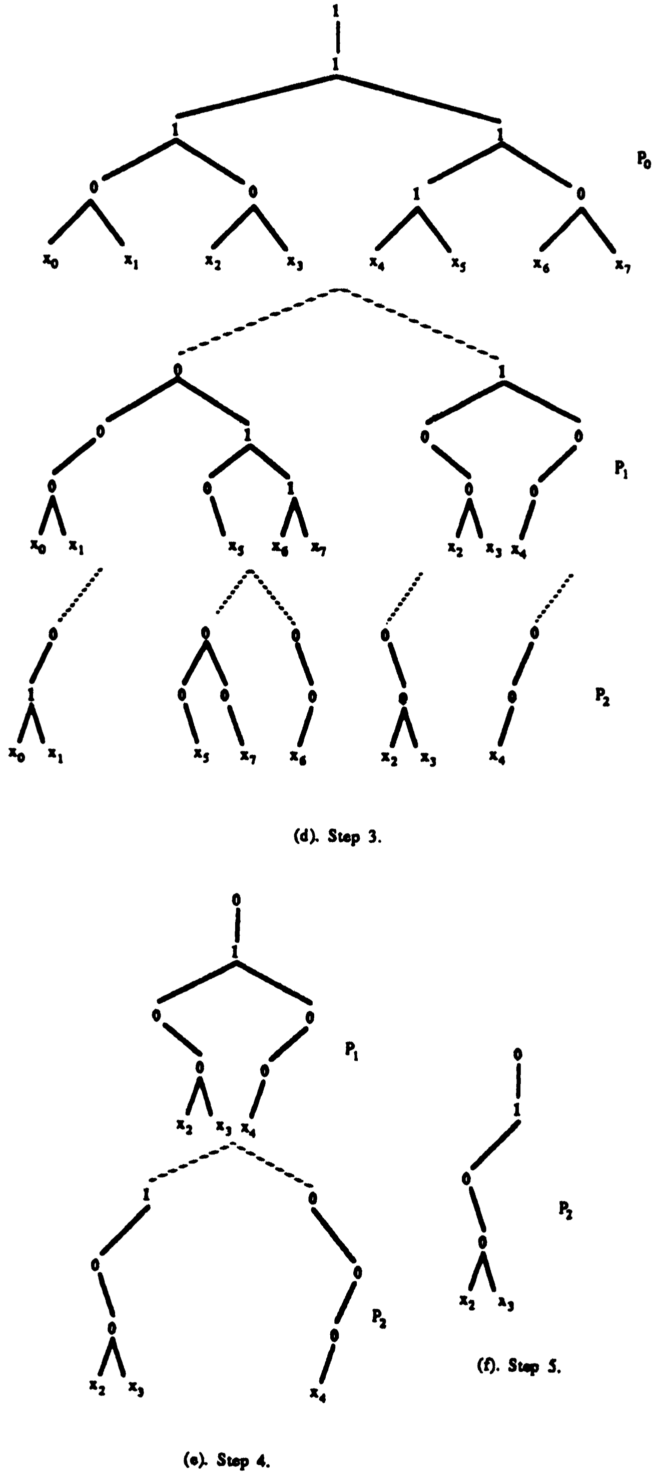

0 0 1 0
x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7
P_0
1 1
0 0 1 0
x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7
P_0
0 0 0 0 1
x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7
P_1
1
1 1
0 0 1 0
x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7
P_0
0 0 0 1
0 0 0 0 1
x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7
P_1
1 0 0 0 0 0
x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7
P_2

(a). Step 0.

(b). Step 1.

(c). Step 2.

FIG. 6. *An execution of MRV-Tree. Darkened lines in boldface are random-variable trees. Dotted lines are bit-pipeline trees.*



(d). Step 3.

(e). Step 4.

(f). Step 5.

FIG. 6. (*Cont.*)

Example. Variables are distributed into the BD sets as shown below.

Step 0:
$x_0, x_1 \in BD(0,0,0,\epsilon)$;
$x_2, x_3 \in BD(0,0,1,\epsilon)$;
$x_4, x_5 \in BD(0,0,2,\epsilon)$;
$x_6, x_7 \in BD(0,0,3,\epsilon)$.

Step 1:
$x_0, x_1, x_2, x_3 \in BD(0,1,0,\epsilon)$;
$x_4, x_5, x_6, x_7 \in BD(0,1,1,\epsilon)$;
$x_0, x_1 \in BD(1,0,0,0)$;
$x_2, x_3 \in BD(1,0,1,0)$;
$x_4 \in BD(1,0,2,0)$;
$x_5 \in BD(1,0,2,1)$;
$x_6, x_7 \in BD(1,0,3,0)$.

Step 2:
$x_0, x_1, x_2, x_3,$
$x_4, x_5, x_6, x_7 \in BD(0,2,0,\epsilon)$;
$x_0, x_1 \in BD(1,1,0,0)$;
$x_2, x_3 \in BD(1,1,0,1)$;
$x_4 \in BD(1,1,1,0)$;
$x_5, x_6, x_7 \in BD(1,1,1,1)$;
$x_0, x_1 \in BD(2,0,0,00)$;
$x_2, x_3 \in BD(2,0,1,00)$;
$x_4 \in BD(2,0,2,00)$;
$x_5 \in BD(2,0,2,10)$;
$x_6 \in BD(2,0,3,00)$;
$x_7 \in BD(2,0,3,01)$.

Step 3:
$x_0, x_1, x_2, x_3,$
$x_4, x_5, x_6, x_7 \in BD(0,3,0,\epsilon)$;
$x_0, x_1, x_5, x_6, x_7 \in BD(1,2,0,0)$;
$x_2, x_3, x_4 \in BD(1,2,0,1)$;
$x_0, x_1 \in BD(2,1,0,00)$;
$x_5, x_7 \in BD(2,1,0,10)$;
$x_6 \in BD(2,1,0,11)$;
$x_2, x_3 \in BD(2,1,1,00)$;
$x_4 \in BD(2,1,1,10)$.

Step 4:
$x_2, x_3, x_4 \in BD(1,3,0,0)$;
$x_2, x_3 \in BD(2,2,0,00)$;
$x_4 \in BD(2,2,0,01)$.

Step 5:
$x_2, x_3 \in BD(2,3,0,00)$.

Now consider GPC functions of the form $Y_i(x_i)$ and $Y_{i,j}(x_i, x_j)$ except the functions in the vertex-cluster term. Our algorithm will distribute these functions into sets $BDF(u,t,i',j')$ by the execution of MRV-Tree, where $BDF(u,t,i',j')$ is essentially the BD set except it is for functions. $Y_{i,j}$ is in $BDF(u,t,i',j')$ iff both x_i and x_j are in $BD(u,t,i',j')$, $\max\{k|$ (the kth bit of $i\ XOR\ j) = 1\} = t$, $|x_i| > u$, and $|x_j| > u$, where XOR is the bitwise exclusive-or operation, with the exception that all functions belong to $BDF(u, \log n, 0, j')$ for some j'. The condition $\max\{k|$ (the kth bit of $i\ XOR\ j) = 1\} = t$ ensures that x_i and x_j are in different subtree of the tree rooted at $T_u^{(t)}[i'][j']$. The conditions $|x_i| > u$ and $|x_j| > u$ ensure that x_i and x_j are still valid. The algorithm for the GPC functions for P_u is shown below.

```
PROCEDURE FUNCTIONS
  begin
    Step $t$ ($0 \le t < u$):
      (*Functions in $BDF(0, t, i', \Lambda)$ reach depth 0 of the bit-pipeline tree for level $t$.*)
      Wait for the pipeline to be filled;
    Step $u + t$ ($0 \le t < \log n$):
      (*Let $S = BDF(u, t, i', j')$.*)
      if $S$ is not empty then
        begin
        for each GPC function $Y_{i,j}(x_i, x_j) \in S$
            compute the BPC function $Y_{i_u,j_u}(x_{i_u}, x_{j_u})$ with conditional bits set to all 0's;

        (*To fix the random bit in $T_u^{(t)}[i'][j']$,*)
        $T_u^{(t)}[i'][j'] := 0$;
        $F_0 := \sum_{Y_{i,j} \in S} Y_{i_u,j_u}(\Psi(x_i, T_u^{(t)}[i'][j']), \Psi(x_j, T_u^{(t)}[i'][j']))$
            $+ \sum_{Y_{i,j} \in S} Y_{i_u,j_u}(\Psi(x_i, T_u^{(t)}[i'][j']) \oplus 1, \Psi(x_j, T_u^{(t)}[i'][j']) \oplus 1) + VC$;
        (*$VC$ is the function value obtained for functions in the vertex-cluster term. We shall explain how to compute it later. $\oplus$ is the exclusive-or operation.*)

        $T_u^{(t)}[i'][j'] := 1$;
        $F_1 := \sum_{Y_{i,j} \in S} Y_{i_u,j_u}(\Psi(x_i, T_u^{(t)}[i'][j']), \Psi(x_j, T_u^{(t)}[i'][j']))$
            $+ \sum_{Y_{i,j} \in S} Y_{i_u,j_u}(\Psi(x_i, T_u^{(t)}[i'][j']) \oplus 1, \Psi(x_j, T_u^{(t)}[i'][j']) \oplus 1) + VC$;

        if $F_0 \geq F_1$ then $T_u^{(t)}[i'][j'] := 0$
        else $T_u^{(t)}[i'][j'] := 1$;
        (*The random bit is fixed.*)

        (*To decide whether $Y_{i,j}$ should remain in the pipeline,*)
        for each $Y_{i,j} \in S$
            begin
              if $\Psi(x_i, T_u^{(t)}[i'][j']) \neq \Psi(x_j, T_u^{(t)}[i'][j'])$ then remove $Y_{i,j}$;
              (*$Y_{i,j}$ is a zero function in the remaining computation of $P_u$ and also a zero function in $P_v$, $v > u$.*)

              if$(\Psi(x_i, T_u^{(t)}[i'][j']) = \Psi(x_j, T_u^{(t)}[i'][j'])) \wedge (|x_i| \geq u+1) \wedge (|x_j| \geq u+1)$ then
                  (*Let $b = \Psi(x_i, T_u^{(t)}[i'][j'])$.*)
                  put $Y_{i,j}$ into $BDF(u+1, t, i', j'b)$; (*$Y_{i,j}$ is to be processed in $Y_{u+1}$.*)

            end
    end
Step $u + \log n$:
  if $S = BDF(u, \log n, 0, j')$ is not empty then
  (*$S$ is the only set left for this step.*)
    begin
        for each GPC function $Y_{i,j}(x_i, x_j)$ $(Y_i(x_i)) \in S$
            compute the BPC function $Y_{i_u,j_u}(x_{i_u}, x_{j_u})$ $(Y_{i_u}(x_{i_u}))$ with conditional bits set to all 0's;

        (*To fix the random bit in $T_u^{(\log n)}[0][j']$,*)
        $T_u^{(\log n)}[0][j'] := 0$;
        $F_0 := \sum_{Y_{i,j} \in S} Y_{i_u,j_u}(\Psi(x_i, T_u^{(\log n)}[0][j']), \Psi(x_j, T_u^{(\log n)}[0][j']))$
            $+ \sum_{Y_i \in S} Y_{i_u}(\Psi(x_i, T_u^{(\log n)}[0][j'])) + VC$;

        $T_u^{(\log n)}[0][j'] := 1$;
        $F_1 := \sum_{Y_{i,j} \in S} Y_{i_u,j_u}(\Psi(x_i, T_u^{(\log n)}[0][j']), \Psi(x_j, T_u^{(\log n)}[0][j']))$
```

```
            $+ \sum_{Y_i \in S} Y_{i_u}(\Psi(x_i, T_u^{(\log n)}[0][j'])) + VC$;

        if $F_0 \geq F_1$ then $T_u^{(\log n)}[0][j'] := 0$
        else $T_u^{(\log n)}[0][j'] := 1$;
        (*The random bit is fixed.*)

        (*To decide whether $T_{i,j}$ should remain in the pipeline,*)
        for each $Y_{i,j} \in S$
            begin
                (*Let $b = T_u^{(\log n)}[0][j']$). $\Psi(x_i, T_u^{(\log n)}[0][j'])$ and $\Psi(x_j, T_u^{(\log n)}[0][j'])$ must be equal here.*)
                if $(\Psi(x_i, T_u^{(\log n)}[0][j']) = \Psi(x_j, T_u^{(\log n)}[0][j']) = 0) \wedge ((|x_i| \geq u + 1) \vee (|x_j| \geq u + 1))$
                then
                    put $Y_{i,j}$ into $BDF(u + 1, \log n, 0, j'b)$;
                else remove $Y_{i,j}$;
            end

        (*To decide whether $Y_i$ should remain in the pipeline,*)
        for each $Y_i \in S$
            begin
                (*Let $b = T_u^{(\log n)}[0][j']$). *)
                if $(\Psi(x_i, T_u^{(\log n)}[0][j']) = 0) \wedge (|x_i| \geq u + 1)$ then
                    put $Y_i$ into $BDF(u + 1, \log n, 0, j'b)$;
                else remove $Y_i$;
            end
    end
end
```

The functions being evaluated can also be viewed as being pipelined through the derandomization trees.

There are $O(\log n)$ steps in MRV-Tree and FUNCTIONS, each step takes $O(\log n)$ time and $O((m + n)\log n)$ processors.

Now we describe how the functions in the vertex-cluster term are evaluated. Each function $Y_{i,j}(x_i, x_j)$ in the vertex-cluster term is defined as $Y_{i,j}(x_i, x_j) = -1$ if the first $|x_i|$ bits of x_i are 0's and the first $|x_j|$ bits of x_j are 0's, and otherwise as $Y_{i,j}(x_i, x_j) = 0$. Let $l(i) = |x_i| - u$. Then $Y_{i_u, j_u}(\Lambda, \Lambda)(0^u, 0^u) = -1/2^{l(i)+l(j)}$ and $Y_{i_u, j_u}(0, 0)(0^u, 0^u) = -1/2^{l(i)+l(j)-2}$ if $|x_i| > u$ and $|x_j| > u$. Procedure MRV-Tree is executed in parallel for each cluster $C(v)$ to build an *evaluation tree* $TC(v)$ for $C(v)$. An evaluation tree is similar to the random-variable tree. The difference between the random-variable tree and $TC(v)$ is that the leaves of $TC(v)$ consist of variables from $C(v)$. Let $r = T_{u,v}^{(t)}[i'][j']$ be the root of a subtree T' in $TC(v)$ which is to be constructed in the current step. Let L and R be the left and right subtrees of T', respectively. Let r_L and r_R be the roots of L and R, respectively. At the beginning of the current step, L and R have already been constructed. Random variables in the interior nodes of L and R have been fixed. Define $M(x, b) = \sum_{\Psi(i,x)=b} \frac{1}{2^{l(i)}}$, where i's are leaves in the subtree rooted at x. At the beginning of the current step, $M(r_L, b)$ and $M(r_R, b)$, $b = 0, 1$, have already been computed and associated with r_L and r_R, respectively. During the current step, r_L is made the left child of r and r_R is made the right child of r. Now r is tentatively set to 0 and 1 to obtain the value VC for fixing r in procedure FUNCTIONS. We first compute $VC(v, r)$ for each

v. $VC(v, r) = 2\sum_{b=0}^{1} M(r_L, b)M(r_R, b \oplus r)$, where $\oplus$ is the exclusive-or operation. The VC value used in procedure FUNCTIONS is $-\sum_{\{v \mid T_{u,v}^{(t)}[i'][j'] \text{ is not empty}\}} \frac{d(i)}{2} VC(v, T_{u,v}^{(t)}[i'][j'])$. After setting r, we obtain an updated value for $M(r, b)$ as $M(r, b) = M(r_L, b) + M(r_R, b \oplus r)$. If T' has only one subtree, then $VC(v, r) = 0$ and $M(r, b)$ need to be computed after r is set.

The above paragraph shows that we need only spend $O(T_{VC})$ operations for evaluating VC for all vertex clusters in a BPC problem, where T_{VC} is the total number of tree nodes of all evaluation trees. T_{VC} is $O(m \log n)$ because there are a total of $O(m)$ leaves and some nodes in the evaluation trees have one child.

We briefly describe the data structure for the algorithm. We build the random-variable tree and evaluation trees for P_0. Nodes $T_0^{(t)}[i][\Lambda]$ in the random-variable tree and nodes $T_{0,v}^{(t)}[i][\Lambda]$ in the evaluation trees and functions in $BDF(0, t, i, \Lambda)$ are sorted by the pair (t, i). This is done only once and takes $O(\log n)$ time with $O(m + n)$ processors [AKS], [C]. As the computation proceeds, the random-variable tree and each evaluation tree will split into several trees; each BDF set will split into several sets, one for each distinct conditional bit pattern. A BDF set in P_u can split into at most two in P_{u+1}. Since we allow $O(\log n)$ time for each step, we can allocate memory for the new level to be built in the evaluation trees. We use pointers to keep track of the bit-pipeline trees and the evaluation trees. The nodes and functions in the same BD and BDF sets (indexed by the same (u, t, i', j')) should be arranged to occupy consecutive memory cells to facilitate the computation of F_0 and F_1 in FUNCTIONS. These operations can be done in $O(\log n)$ time using $O((m + n)/\log n)$ processors.

It is now straightforward to verify that our algorithm for solving a GPC problem takes $O(\log^2 n)$ time, $O(\log n)$ time for each of the $O(\log n)$ steps. We note that in each step for each BPC problem we have used $O(m + n)$ processors. This can be reduced to $O((m + n)/\log n)$ processors because in each step, $O(m + n)$ operations are performed for each BPC problem. They can be done in $O(\log n)$ time using $O((m + n)/\log n)$ processors. Since we have $O(\log n)$ BPC problems, we need only $O(m + n)$ processors to achieve time complexity $O(\log^2 n)$ for solving one GPC problem.

We use $O((m + n)/\log^{0.5} n)$ processors to solve the first $O(\log^{0.5} n)$ GPC problems in the maximal independent set problem. Recall that the execution of a GPC algorithm will reduce the size of the graph by a constant fraction. For the first $O(\log \log n)$ GPC problems, the time complexity is $O(\sum_{i=1}^{O(\log \log n)} \log^{2.5} n/c^i) = O(\log^{2.5} n)$, where $c > 1$ is a constant. In the ith GPC problem, we solve $O(c^i \log^{0.5} n)$ BPC problems in a batch, incurring $O(\log^2 n)$ time for one batch and $O(\log^{2.5} n/c^i)$ time for the $O(\log^{0.5} n/c^i)$ batches. The time complexity for the remaining GPC problems is $O(\sum_{i=O(\log \log n)}^{\log^{0.5} n} \log^2 n) = O(\log^{2.5} n)$.

5.3. The speedup stage. The input graph here is the output graph from the initial stage. The speedup stage consists of the rest of the GPC problems.

We have to reduce the time complexity for solving one GPC problem to under $O(\log^2 n)$ in order to obtain an $o(\log^3 n)$ algorithm for the maximal independent set problem. After the initial stage, we have a small-size problem and we have extra processor power to help us speed up the algorithm.

We redesign the random-variable tree T for a BPC problem. We use the design as shown in Fig. 3(c) in §2. There are $S = \lceil(\log n + 1)/a\rceil$ blocks in T, where a is a parameter.

We note that the design of T incorporates design techniques from both [H], [HI] and [L2], [L3]. The advantage of Han and Igarashi's design [H], [HI] is that random bits can be fixed independently if these bits are at the same level of T. The advantage of Luby's design is that there are fewer random bits in T, which is desirable in the speedup stage of our algorithm for the maximal independent set problem.

LEMMA 6. *If all random variables in the interior nodes of a proper subtree T' of T are fixed, the random variables x_j at the leaves of T' can only assume two different patterns.*

Proof. This is because the random variables from the root of T to the parent of the root of T' are common to all x_j's at the leaves of T'. □

In fact, we have implicitly used this lemma in constructing the bit-pipeline tree in the design of our GPC algorithm and in procedure RV-Tree.

The q BPC problems in a GPC problem are divided into $b = q/a$ teams (without loss of generality, assuming it is an integer). Team i, $0 \leq i < b$, has a BPC problems. Let J_w be the wth team. The algorithm for fixing the random variables for J_w can be expressed as follows.

Step t $(0 \leq t < w)$: Wait for the pipeline to be filled.
Step $t + w$ $(0 \leq t < S)$: Fix random variables in block t in random-variable forests for J_w.

Each step will be executed in $O(\log n)$ time. Since there are $O(b + S)$ steps, the time complexity is $O(\log^2 n/a)$ for the above algorithm since $q = O(\log n)$.

For a graph with m edges and n vertices, to fix random bits in block 0 for P_0, we need $2^a(m + n)$ processors to enumerate all possible 2^a bit patterns for the a bits in block 0. To fix the bits in block 0 for P_v, $v < a$, we need $2^{a(v+1)}$ patterns to enumerate all possible $a(v + 1)$ bits in block 0 for P_u, $u \leq v$. For each of the $2^{a(v+1)}$ patterns, there are 2^v conditional-bit patterns. Thus we need $c^{a^2}(m + n)$ processors for team 0 for a suitable constant c. Although the input to each team may have many conditional-bit patterns, it contains at most $O(m + n)$ random-variable trees (in the input random-variable forest). We need keep working for only those conditional-bit patterns which are not associated with empty trees. Thus the number of processors needed for each team is the same because when team J_w is working on block i, the bits in block i have already been fixed for teams J_u, $u < w$, and because we keep only nonzero functions. The situation here is similar to the situation in the initial stage. Thus the total number of processors we need for solving one GPC problem in time $O(\log^2 n/a)$ is $c^{a^2}(m+n)\log n/a = O(c^{a^2}(m+n)\log n)$. We conclude that one GPC problem can be solved in time $O(\log^2 n/\sqrt{k})$ with $O(c^k(m + n)\log n)$ processors. Therefore, the time complexity for the speedup stage is $O(\sum_{k=1}^{\log n}(\log^2 n/\sqrt{k})) = O(\log^{2.5} n)$.

THEOREM 6. *There is an EREW PRAM algorithm for the maximal independent set problem with time complexity $O(\log^{2.5} n)$ using $O((m + n)/\log^{0.5} n)$ processors.*

We shall call this algorithm MAX.

5.4. Further improvement. To reduce the processor complexity by another factor of $\log n$ on the CREW model, we need only work on the first $O(\log\log n)$ GPC problems. These GPC problems belong to the initial stage.

Consider the first GPC problem. At the beginning of the GPC algorithm, all GPC functions (in $BDF(0, t, i, \Lambda)$), nodes in the random-variable tree (in $T_0^{(t)}[i][\Lambda]$) and nodes in the evaluation trees (in $T_{0,v}^{(t)}[i][\Lambda]$) will be sorted by the parameter (t, i). This takes $O(m\log n/p+\log n)$ time with p processors. A GPC function f will be passed down the bit-pipeline tree in the procedure FUNCTIONS. At each depth of the bit-pipeline tree, f is involved in a constant number of operations. Thus each GPC function will account for $O(\log n)$ operations, giving a total of $O(m\log n)$ operations. This can be done in time $O(m\log n/p + \log^2 n)$ with p processors. The nodes in the random-variable tree and the nodes in an evaluation tree, as they pass down the bit-pipeline tree, can be decomposed into several random-variable trees and evaluation trees, one for each conditional-bit pattern. Each leaf in these trees can be involved in $O(\log n)$ operations in a BPC problem and therefore $O(\log^2 n)$ operations in the GPC problem. This gives time $O(m\log^2 n/p + \log^2 n)$. On the CREW PRAM, we can avoid evaluating nodes in a evaluation tree which has only one child. As long as we only evaluate nodes in the evaluation trees which have two children, the number of operations for evaluating the nodes in an evaluation tree is proportional to the number of leaves in the tree. This helps to cut the time for evaluating the evaluation trees to $O(m/p + \log^2 n)$ for a BPC problem and to $O(m \log n/p + \log^2 n)$ for the GPC problem. However, a node v at level l in an evaluation tree could have its parent $p(v)$ at level $l + c$ with $c > 1$, because now we require that $p(v)$ have two children. When $p(v)$ is evaluated, we need the value $\Psi(v, p(v))$. In order to obtain this value, we keep updated $\Psi(v, w)$ for all leaves v in a random-variable tree and the current node w. The $\Psi(v, w)$ value for the n leaves in the random-variable forest for a BPC problem will be updated immediately after the random variables at each level are fixed. This takes $O(n \log n/p)$ time for a BPC problem and $O(n \log^2 n/p)$ time for the GPC problem. In summary, the first GPC problem can be solved in time $O(m \log n/p + n \log^2 n/p + \log^2 n)$ with p processors. If $m > n \log n$, the time will become $O(m \log n/p + \log^2 n)$.

One might argue that since the evaluation trees are built bottom-up, if a node is not checked one cannot know whether that node has one or two children. The answer is that we cannot avoid checking whether a node r in an evaluation tree of P_u has one or two children. But if we know r has one child, we can avoid checking r's descendants in the bit-pipeline tree, i.e., those nodes in P_v, $v > u$, which are descendants of r in the bit-pipeline tree.

Our modified algorithm for the GPC problem will first check whether $m > n \log n$. If $m \leq n \log n$ we first construct G' induced by vertices in V with degree no greater than $\log n$. We then solve the maximal independent set problem for G' in time $O(m \log n/p + \log^2 n)$, using an algorithm to be described later. Now the remaining graph can be viewed as satisfying $m > n \log n$, and the rest of the computation takes $O(m \log n/p + \log^2 n)$ time as explained above. We therefore achieve time $O(m \log n/p + \log^2 n \log\log n)$ for the first $O(\log\log n)$ GPC problems. The remaining graph can now be solved by MAX.

We now describe an algorithm for finding a maximal independent set for a graph satisfying $\Delta = O(\log n)$. This algorithm is obtained by using a modified version of our $\Delta + 1$ vertex-coloring algorithm. We first color the graph with $\Delta + 1$ colors and then find a maximal independent set by sequencing through these colors.

LEMMA 7 [HI]. *A BPC problem can be solved by first sorting the input BPC functions into the file-major indexing, which takes $O(m \log n/p + \log n)$ time, and then building the derandomization tree and derandomizing the random variables, which takes $O(m/p + \log n)$ time, where p is the number of processors used.*

The reason that the computation for a BPC problem except the sorting step takes $O(m/p + \log n)$ time is that the derandomization process can be formulated [HI] as a tree-contraction process [MR]. Note that in order to establish Lemma 7, the derandomization tree D should take the form such that each interior node of D must have at least two children [HI].

LEMMA 8. *The $\Delta + 1$ vertex-coloring problem can be solved in time $O(m \log n/p + \log n(\log\log n)^2)$ using p processors on the CREW PRAM if $\Delta = O(\log n)$.*

Proof. Since $\Delta = O(\log n)$, one GPC problem now contains only $O(\log\log n)$ BPC problems. Since the derandomization trees for all the BPC problems in a GPC problem are the same, we need only build one derandomization tree and then make $O(\log\log n)$ copies of the tree. The time complexity for building the derandomization trees in the GPC algorithm is $O(m \log n/p + \log n)$. The time complexity for building the tables is $O(m \log\log n/p + \log n)$ for a BPC problem, because $\Delta = O(\log n)$, and $O(m(\log\log n)^2/p + \log n \log\log n)$ for the GPC problem. The time complexity for the rest of the computation in the GPC algorithm is $O(m \log\log n/p + \log n \log\log n)$ using p processors because the BPC problems are solved sequentially. Thus the first $O(\log\log n)$ GPC problems can be solved in time

$$O\left(\left(\sum_{i=1}^{O(\log\log n)} \frac{m \log n}{c^i p}\right) + \log n(\log\log n)^2\right) = O(m \log n/p + \log n(\log\log n)^2)$$

using p processors. Now the graph has size $O(m/\log^2 n)$. It can be colored using the algorithm in §4. Note again that each GPC problem has only $O(\log\log n)$ BPC problems. □

THEOREM 7. *There is a CREW PRAM algorithm for the maximal independent set problem with time complexity $O(\log^{2.5} n)$ using $O((m+n)/\log^{1.5} n)$ processors.*

The dominating operations in each step of our maximal independent set algorithm are memory allocation and summation. These operations can be done in time $O(\log n/\log\log n)$ on the CRCW PRAM [P], [Re], [CV]. Therefore, we have the following corollary.

COROLLARY. *There is a CRCW PRAM algorithm for the maximal independent set problem with time complexity $O(\log^{2.5} n/\log\log n)$ using $O((m+n)\log\log n/\log^{1.5} n)$ processors.*

6. Maximal matching. Let $N(M) = \{(i,k) \in E,\ (k,j) \in E \mid \exists (i,j) \in M\}$. A maximal matching can be found by repeatedly finding a matching M and removing $M \cup N(M)$ from the graph.

We adapt Luby's work [L3] to show that after an execution of the GPC procedure a constant fraction of the edges will be reduced.

Let k_i be such that $2^{k_i-1} < 4d(i) \leq 2^{k_i}$. Let $q = \max\{k_i | i \in V\}$. Let $\vec{x} =< x_{ij} \in \{0,1\}^q, (i,j) \in E\}$. The length $|x_{ij}|$ of x_{ij} is defined to be $\max\{k_i, k_j\}$. Define

$$Y_{ij}(x_{ij}) = \begin{cases} 1 & \text{if } x_{ij}(|x_{ij}|-1)\cdots x_{ij}(0) = 0^{|x_{ij}|}, \\ 0 & \text{otherwise.} \end{cases}$$

$$Y_{ij,i'j'}(x_{ij}, x_{i'j'}) = -Y_{ij}(x_{ij})Y_{i'j'}(x_{i'j'}),$$

$$B(\vec{x}) = \sum_{i\in V} \frac{d(i)}{2} \left(\sum_{j\in adj(i)} \left(Y_{ij}(x_{ij}) + \sum_{k\in adj(j), k\neq i} Y_{ij,jk}(x_{ij}, x_{jk}) \right) + \sum_{j,k\in adj(i), j\neq k} Y_{ij,ik}(x_{ij}, x_{ik}) \right),$$

where $x_{ij}(p)$ is the pth bit of x_{ij}.

Function B sets a lower bound on the number of edges deleted from the graph [L3] should edge (i,j) be tentatively labeled as an edge in the matching set if $x_{ij} = (0 \cup 1)^{q-|x_{ij}|}0^{|x_{ij}|}$. The following lemma can be proven by following Luby's proof for Theorem 1 in [L1].

LEMMA 9. *$E[B] \geq |E|/c$ for a constant $c > 0$.*

Function B can be written as

$$B(\vec{x}) = \sum_{(i,j)\in E} \frac{d(i)+d(j)}{2} Y_{ij}(x_{ij}) + \sum_{j\in V} \sum_{i,k\in adj(j), i\neq k} \frac{d(i)}{2} Y_{ij,jk}(x_{ij}, x_{jk})$$

$$+ \sum_{i\in V} \frac{d(i)}{2} \sum_{j,k\in adj(i), j\neq k} Y_{ij,ik}(x_{ij}, x_{ik})$$

By using the same technique as in §5, we can obtain a CREW algorithm for the maximal-matching problem with time complexity $O(\log^2 n)$ using $O(M(n))$ processors. The details of this algorithm are omitted here.

There are two cluster terms in function B. We only need explain how to evaluate the cluster term $\sum_{j\in V} \sum_{i,k\in adj(j), i\neq k} \frac{d(i)}{2} Y_{ij,jk}(x_{ij}, x_{jk})$. The rest of the functions can be computed as we have done for the maximal independent set problem in §5.

Again we build an evaluation tree $TC(v)$ for each cluster $C(v)$ in the cluster term. Let $l(ij) = |x_{ij}| - u$. Let $r = T^{(t)}_{u,v}[i'][j']$ be the root of a subtree T' in $TC(v)$ which is to be constructed in the current step. Let L and R be the left and right subtrees of T', respectively. Let r_L and r_R be the roots of L and R, respectively. At the beginning of the current step, L and R have already been constructed. Random variables in the interior nodes of L and R have been fixed. Define $M(x,b) = \sum_{\Psi(ij,x)=b} \frac{1}{2^{l(ij)}}$. Define $N(x,b) = \sum_{\Psi(ij,x)=b} \frac{d(i)}{2} \frac{1}{2^{l(ij)}}$. At the beginning of the current step, $M(r_L, b)$, $M(r_R, b)$, $N(r_L, b)$, and $N(r_R, b)$, $b = 0, 1$, have already been computed and associated with r_L and r_R, respectively. During the current step, r_L is made the left child of r and r_R is made the right child of r. Now r is tentatively set to 0 and 1 to obtain value VC for fixing r. We first compute $VC(v, r)$ for each v. $VC(v,r) = \sum_{b=0}^{1}(N(r_L, b)M(r_R, b \oplus r) + M(r_L, b)N(r_R, b \oplus r))$. The VC value is $-\sum_{\{v | T^{(t)}_{u,v}[i'][j'] \text{ is not empty}\}} VC(v, T^{(t)}_{u,v}[i'][j'])$. After setting r, we obtain updated values $M(r,b)$ and $N(r,b)$ as $M(r,b) = M(r_L, b) + M(r_R, b \oplus r), N(r,b) = N(r_L, b) + N(r_R, b \oplus r)$.

Since this computation does not require more processors, we have the following theorem.

THEOREM 8. *There is an EREW PRAM algorithm for the maximal matching problem with time complexity $O(\log^{2.5} n)$ using $O((m+n)/\log^{0.5} n)$ processors.*

For the maximal-matching problem, we cannot remove another factor of $\log n$ from the processor complexity as we did for the maximal independent set problem because there are $O(m)$ leaves in the random-variable trees of a BPC problem, while there are only $O(n)$ leaves in the maximal independent set problem.

Again in the CRCW PRAM algorithm a factor of $\log \log n$ can be taken out from the time complexity and put into the processor complexity.

Acknowledgments. The author wishes to thank the anonymous referees for their careful reviewing and helpful comments on the manuscript. The author also thanks Professor Richard Cole for making several good suggestions about the paper.

REFERENCES

[AKS] M. AJTAI, J. KOMLÓS, AND E. SZEMERÉDI, *An $O(N \log N)$ sorting network*, in Proc. 15th ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1983, pp. 1-9.

[ABI] N. ALON, L. BABAI, AND A. ITAI, *A fast and simple randomized parallel algorithm for the maximal independent set problem*, J. Algorithms, 7 (1986), pp. 567-583.

[BR] B. BERGER AND J. ROMPEL, *Simulating* $(\log^c n)$*-wise independence*, in NC Proc. 30th Symposium on Foundations of Computer Science, IEEE Press, Piscataway, NJ, 1989, pp. 2-7.

[BRS] B. BERGER, J. ROMPEL, AND P. SHOR, *Efficient NC algorithms for set cover with applications to learning and geometry*, in Proc. 30th Symposium on Foundations of Computer Science, IEEE Press, Piscataway, NJ, 1989, pp. 54-59.

[C] R. COLE, *Parallel merge sort*, in Proc. 27th Symposium on Foundations of Computer Science, IEEE Press, Piscataway, NJ, 1986, pp. 511-516.

[Co] S. COOK, *A taxonomy of problems with fast parallel algorithms*, Inform. and Control, 64 (1985), pp. 2-22.

[CV] R. COLE AND U. VISHKIN, *Approximate and exact parallel scheduling with applications to list, tree and graph problems*, in Proc. 27th Symposium on Foundations of Computer Science, IEEE Press, Piscataway, NJ, 1986, pp. 478-491.

[CW] D. COPPERSMITH AND S. WINOGRAD, *Matrix multiplication via arithmetic progressions*, in Proc. 19th Annnual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1987, pp. 1-6.

[FW] S. FORTUNE AND J. WYLLIE, *Parallelism in random access machines*, in Proc. 10th ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1978, pp. 114-118.

[GS1] M. GOLDBERG AND T. SPENCER, *A new parallel algorithm for the maximal independent set problem*, SIAM J. Comput., 18 (1989), pp. 419-427.

[GS2] ———, *Constructing a maximal independent set in parallel*, SIAM J. Discrete Math., 2 (1989), pp. 322-328.

[HCD] T. Hagerup, M. Chrobak, and K. Diks, *Optimal parallel 5-coloring of planar graphs*, SIAM J. Comput., 18 (1989), pp. 288-300.

[H] Y. Han, *A parallel algorithm for the PROFIT/COST problem*, in Proc. 1991 International Conference on Parallel Processing, 1991, pp. 103-112.

[HI] Y. Han and Y. Igarashi, *Derandomization by exploiting redundancy and mutual independence*, Lecture Notes in Comput. Sci., 450 (1990), pp. 328-337.

[II] A. Israeli and A. Itai, *A fast and simple randomized parallel algorithm for maximal matching*, Tech. report, Computer Science Deptartment, Technion, Haifa, Israel, 1984.

[IS] A. Israeli and Y. Shiloach, *An improved parallel algorithm for maximal matching*, Inform. Process. Lett., 22 (1986), pp. 57-60.

[KW] R. Karp and A. Wigderson, *A fast parallel algorithm for the maximal independent set problem*, J. Assoc. Comput. Mach., 32 (1985), pp. 762-773.

[L1] M. Luby, *A simple parallel algorithm for the maximal independent set problem*, SIAM J. Comput., 15 (1986), pp. 1036-1053.

[L2] ———, *Removing randomness in parallel computation without a processor penalty*, in Proc. 29th Symposium on Foundations of Computer Science, IEEE Press, Piscataway, NY, 1988, pp. 162-173.

[L3] ———, *Removing randomness in parallel computation without a processor penalty*, Tech. report, 89-044, International Computer Science Institute, Berkeley, CA, J. Comput. System Sci., 47 (1993), pp. 250-286.

[MR] G. L. Miller and J. H. Reif, *Parallel tree contraction and its application*, in Proc. 26th Symposium on Foundations of Computer Science, IEEE Press, Piscataway, NJ, 1985, pp. 478-489.

[MNN] R. Motwani, J. Naor, and M. Naor, *The probabilistic method yields deterministic parallel algorithms*, in Proc. 30th Symposium on Foundations of Computer Science, IEEE Press, Piscataway, NJ, 1989, pp. 8-13.

[PSZ] G. Pantziou, P. Spirakis, and C. Zaroliagis, *Fast parallel approximations of the maximum weighted cut problem through derandomization*, Lecture Notes in Comput. Sci., 405 (1989), pp. 20-29.

[P] I. Parberry, *On the time required to sum n semigroup elements on a parallel machine with simultaneous write*, Lecture Notes on Comput. Sci., 227, pp. 296-304.

[Rag] P. Raghavan, *Probabilistic construction of deterministic algorithms: Approximating packing integer programs*, J. Comput. System Sci., 37 (1988), pp. 130-143.

[Re] J. H. Reif, *An optimal parallel algorithm for integer sorting*, in Proc. 26th Symposium on Foundations of Computer Sci., IEEE Press, Piscataway, NJ, 1985, pp. 291-298.

[Sp] J. Spencer, *Ten Lectures on the Probabilistic Method*, Society for Industrial and Applied Mathematics, Philadelphia, 1987.

SIAM J. COMPUT.
Vol. 25, No. 1, pp. 83–99, February 1996

WEIGHTED MULTIDIMENSIONAL SEARCH AND ITS APPLICATION TO CONVEX OPTIMIZATION*

RICHA AGARWALA† AND DAVID FERNÁNDEZ-BACA‡

Abstract. We present a weighted version of Megiddo's multidimensional search technique and use it to obtain faster algorithms for certain convex optimization problems in $\mathbf{R}^d$, for fixed d. This leads to speed-ups by a factor of $\log^d n$ for applications such as solving the Lagrangian duals of matroidal knapsack problems and of constrained optimum subgraph problems on graphs of bounded tree-width.

Key words. computational geometry, convex optimization, Lagrangian relaxation, multidimensional search

AMS subject classifications. 52B12, 52B30, 52B55, 68P10, 68Q25, 68U05

1. Introduction.

This paper has three main parts. In the first (§2), we present a weighted version of the multidimensional search technique of Megiddo [27], [17], [10]. The second part (§3) discusses the application of our result to a class of convex optimization problems in fixed dimension which were studied earlier by Cohen and Megiddo [14], [15] and in a different context by Aneja and Kabadi [4]. In rough terms, the results in [14], [15], and [4] can be summarized as follows. Suppose that g is a concave function whose domain $\mathcal{Q}$ is a convex subset of $\mathbf{R}^d$ and that g is computable in $O(T)$ time by an algorithm $\mathcal{A}$ that only performs additions, multiplications by constants, copies, and comparisons on intermediate values that depend on the input numbers. Then g can be maximized in $O(T^{d+1})$ time. Cohen and Megiddo go on to show that substantial speed-ups are possible by exploiting whatever parallelism is inherent to algorithm $\mathcal{A}$. Thus, if $\mathcal{A}$ carries out D parallel steps, each of which does at most M comparisons, the running time will be $O((D \log M)^d T)$. By applying weighted multidimensional search and a generalization of Cole's circuit-simulation technique [16], we are able to reduce this to $O((D^d + \log^d M)T)$ in some cases.

Lagrangian relaxation is a source of several problems that fall into the framework described above [4]. This widely used approach is based on the observation that many hard optimization problems are actually easy problems that are complicated by a relatively small set of side constraints. By "pricing out" the bad side constraints into the objective function, one obtains a simpler convex optimization problem whose optimum solution provides good bounds on the optimum value of the original problem. The third part of this paper (§4) explains the application of our results to Lagrangian relaxation problems where the number of bad constraints is fixed. We give two examples of problems where the methods described in §3 give faster algorithms than those of [4], [14]: solving the Lagrangian duals of *matroidal knapsack problems* [11] and of certain constrained optimum subgraph problems on graphs of bounded tree-width.

2. Weighted multidimensional search.

Let us first introduce some notation. Suppose $\Lambda \subseteq \mathbf{R}^d$ is convex and that $h : \mathbf{R}^d \to \mathbf{R}$ is an affine function. Define $\text{sign}_\Lambda(h)$ as

$$\text{sign}_\Lambda(h) = \begin{cases} 0 & \text{if } h(\lambda) = 0 \text{ for some } \lambda \in \Lambda, \\ +1 & \text{if } h(\lambda) > 0 \text{ for all } \lambda \in \Lambda, \\ -1 & \text{if } h(\lambda) < 0 \text{ for all } \lambda \in \Lambda. \end{cases}$$

*Received by the editors December 28, 1992; accepted for publication (in revised form) July 22, 1994. An earlier version of this paper appears in [1].

†Department of Computer Science, Iowa State University, Ames, IA 50011. The research of this author was supported in part by an Iowa State University College of Liberal Arts and Sciences Research Assistantship. Current address: DIMACS (The Center for Discrete Mathematics and Theoretical Computer Science), Rutgers University, Piscataway, NJ 08855 (agarwala@dimacs.rutgers.edu).

‡Department of Computer Science, Iowa State University, Ames, IA 50011 (fernande@cs.iastate.edu). The research of this author was supported in part by the National Science Foundation grants CCR-8909626 and CCR-9211262. This author's work was conducted in part at DIMACS, Rutgers University.

We will write sign for sign_Λ when no confusion can arise. A function h is *resolved* if $\text{sign}_\Lambda(h)$ has been computed. Obviously, if $h(\lambda) = a_0$, $\text{sign}(h)$ can be immediately determined from the sign of a_0.

Suppose we have a set $\mathcal{H}$ of d-dimensional affine functions and an oracle $\mathcal{B}^d$ that can compute $\text{sign}_\Lambda(h)$ for any $h \in \mathcal{H}$. The problem is to resolve every $h \in \mathcal{H}$ using as few oracle calls as possible. The following result is proved in [27], [17], [10].

THEOREM 2.1. *For each fixed $d \geq 0$ there exist positive constants $\beta(d)$ and $\alpha(d)$, $\alpha(d) \leq 1/2$, and an algorithm* SEARCH *such that, given a set $\mathcal{H}$ of affine functions,* SEARCH *either returns an affine function h such that* $\text{sign}_\Lambda(h) = 0$ *or resolves every $h \in \mathcal{H}' \subseteq \mathcal{H}$, where $|\mathcal{H}'| \geq \alpha(d) \cdot |\mathcal{H}|$, by making at most $\beta(d)$ calls to $\mathcal{B}^d$. Furthermore, the work done by* SEARCH *in addition to the oracle calls is $O(|\mathcal{H}|)$.*

In reality, the above references have proofs of this result for the case where Λ is a single point, but the proof extends easily to the case where Λ is a convex set. By repeatedly applying algorithm SEARCH, we can resolve all functions in $\mathcal{H}$ with $O(\log|\mathcal{H}|)$ oracle calls. In this section, we shall prove a weighted version of Theorem 2.1. Let S be a set on which a weight function $w : S \to \mathbf{R}^+$ has been defined. For $S' \subseteq S$, we write $w(S')$ to denote $\sum_{s \in S'} w(s)$. We have the following result.

THEOREM 2.2. *For each $d \geq 0$, there exist constants $\beta(d)$ and $\alpha(d)$, $\alpha \leq 1/2$, and an algorithm* WEIGHTED-SEARCH *with the following property. Given a set $\mathcal{H}$ of affine functions and a weight function $w : \mathcal{H} \to \mathbf{R}^+$,* WEIGHTED-SEARCH *either returns an affine function h such that* $\text{sign}_\Lambda(h) = 0$ *or finds a subset $\mathcal{H}' \subseteq \mathcal{H}$ with $w(\mathcal{H}') \geq \alpha(d) \cdot w(\mathcal{H})$ and resolves every $h \in \mathcal{H}'$ by making at most $\beta(d)$ calls to $\mathcal{B}^d$. Furthermore, the work done by* WEIGHTED-SEARCH *in addition to the oracle calls is $O(|\mathcal{H}|)$.*

The proof of this theorem will require some preliminary results, which are discussed next.

2.1. Preliminaries. Procedure WEIGHTED-SEARCH uses two simple algorithms. The first is MATCH, which, given two sets A and B, attempts to match disjoint subsets of B with elements of A in a "greedy" manner.

```
ALGORITHM MATCH
Input: Sets $A = \{a_1, \ldots, a_{|A|}\}$ and $B = \{b_1, \ldots, b_{|B|}\}$ and a weight function $w$ :
    $A \cup B \to \mathbf{R}^+$.
Output: Either FAILURE or disjoint sets $S_1, \ldots, S_{|A|}$ such that, for each $i$, $S_i = \{a_i\} \cup D_i$
    where $D_i \subseteq B$ and $w(D_i) \geq w(a_i)$.
begin
    $j \leftarrow 1$
    for $i = 1$ to $|A|$ do begin
        $D_i \leftarrow \emptyset$
        while $w(D_i) < w(a_i)$ do begin
            if $j > |B|$ then return FAILURE
            $D_i \leftarrow D_i \cup b_j$
            $j \leftarrow j + 1$
        end
        $S_i \leftarrow \{a_i\} \cup D_i$
    end;
    return $S_1, \ldots, S_{|A|}$
end
```

The running time of this algorithm is clearly $O(|B|)$. We shall say that MATCH *succeeds* if it does not return FAILURE. If MATCH succeeds, then the solution returned obviously satisfies its output conditions. The next lemma gives one scenario in which MATCH always succeeds.

LEMMA 2.3. *If* $\min_{x\in A} w(x) \geq \max_{y\in B} w(y)$ *and* $w(B) \geq 2w(A)$, *then* MATCH *succeeds. Furthermore, each set* $S_i = \{a_i\} \cup D_i$ *returned by* MATCH *satisfies* $w(D_i) < 2w(a_i)$.

Proof. (This is a proof by contradiction.) Suppose the conditions of the lemma hold and that MATCH returns FAILURE. Then there exists a k, $1 \leq k \leq |A|$, such that, at the kth iteration of the **for** loop, MATCH runs out of elements of B to match up with a_k; i.e., $w(D_k) < w(a_k)$ and $B = \cup_{i=1}^{k} D_i$, where $D_1, \ldots, D_k$ are the subsets of B constructed by MATCH up to this point. Thus, $\sum_{j=1}^{k} w(D_j) = w(B)$. For $j = 1, \ldots, k$, let $D_j = \{d_{j1}, d_{j2}, \ldots, d_{jl_j}\}$. By construction, for $1 \leq j \leq k-1$, $w(D_j) - w(d_{jl_j}) < w(a_j)$. Thus, for $1 \leq j \leq k-1$, $w(D_j) < w(a_j) + w(d_{jl_j}) \leq 2w(a_j)$, as $\min_{x\in A} w(x) \geq \max_{y\in B} w(y)$. Together with the above-mentioned fact that $w(D_k) < w(a_k)$, we get that

$$w(B) = \sum_{j=1}^{k} w(D_j) < \sum_{j=1}^{k} 2w(a_j) \leq 2w(A),$$

which is a contradiction. Therefore, MATCH succeeds, and for each set $S_i = \{a_i\} \cup D_i$, $w(D_i) < 2w(a_i)$. □

MATCH is invoked by the following algorithm.

ALGORITHM PAIRING

Input: Sets A, B, a weight function $w : A \cup B \to \mathbf{R}^+$, and a number m such that $W/2 \geq w(B) \geq w(A) \geq (W/2 - m)$, where $W = w(A \cup B) + m$.

Output: $k \geq 0$ disjoint sets $S_1, \ldots, S_k$, and an element e satisfying the following conditions:

(P1) Each S_i has the form $S_i = \{c_i\} \cup D_i$, where $e \neq c_i$, and either
(1) for all i, $e, c_i \in A$ and $D_i \subseteq B$, or
(2) for all i, $e, c_i \in B$ and $D_i \subseteq A$.
(P2) for all i, $2w(c_i) > w(D_i) \geq w(c_i)$.
(P3) $\sum_{i=1}^{k} w(c_i) + w(e) + m \geq W/6$.

Note: In order to break ties between items with equal weights, we assume an arbitrary but fixed ordering among the elements in A and in B. Given any two elements x and y, where either both are in A or both are in B, we will say that x precedes y if $(w(x), x)$ is lexicographically smaller than $(w(y), y)$.

Step 1. Find $a \in A$ and $b \in B$ such that $w(A_1), w(B_1) \leq w(A)/3$ and $w(A_1 \cup \{a\}), w(B_1 \cup \{b\}) \geq w(A)/3$, where $A_1 = \{x \in A : x \text{ precedes } a\}$ and $B_1 = \{x \in B : x \text{ precedes } b\}$. Let $A_2 = A - A_1$ and $B_2 = B - B_1$.

Step 2. If $w(a) \geq w(b)$, do the following steps.

Step 2(a). If $w(a) + m \geq W/6$, then return $k = 0$ and $e = a$.

Step 2(b). Call MATCH with inputs A_1 and B_2. Let $S_1, \ldots, S_{|A_1|}$ be the sets returned by this call. Return $S_1, \ldots, S_{|A_1|}$, and $e = a$.

Step 3. If $w(a) < w(b)$, do the following steps.

Step 3(a). If $w(b) + m \geq W/6$, then return $k = 0$ and $e = b$.

Step 3(b). Call MATCH with inputs B_1 and A_2. Let $S_1, \ldots, S_{|B_1|}$ be the sets returned by this call. Return $S_1, \ldots, S_{|B_1|}$ and $e = b$.

LEMMA 2.4. PAIRING *correctly computes output satisfying conditions* (P1)–(P3).

Proof. If the output is returned in Step 2(a) or Step 3(a), the conditions are trivially satisfied. We now consider Step 2(b); the analysis for Step 3(b) is similar. By construction, $\min_{x\in A_1} w(x) \geq \max_{y\in B_2} w(y)$ and $w(B_2) \geq 2w(A)/3 \geq 2w(A_1)$. Since the conditions of Lemma 2.3 are satisfied, MATCH succeeds and conditions (P1) and (P2) are satisfied. Since MATCH works correctly, we have $\sum_{i=1}^{|A_1|} w(c_i) = w(A_1)$. Therefore, $\sum_{i=1}^{k} w(c_i) + w(e) + m =$

$w(A_1) + w(a) + m$. Because $w(A_1) + w(a) \geq w(A)/3$,

$$\sum_{i=1}^{k} w(c_i) + w(e) + m \geq w(A)/3 + m \geq (w(A) + m)/3.$$

Since $w(A) + m \geq W/2$, we obtain

$$\sum_{i=1}^{k} w(c_i) + w(e) + m \geq W/6$$

as desired. □

PAIRING can be implemented to run in $O(n)$ time, where $n = |A| + |B|$. Step 1 takes $O(n)$ time as elements a and b can be found by repeated median finding [12]. Steps 2 and 3 also take linear time, since MATCH takes linear time.

2.2. The search algorithm. We shall now prove Theorem 2.2. The implementation of WEIGHTED-SEARCH that we propose is an extension of Megiddo's [27] and Dyer's [17] algorithms for unweighted multidimensional search (see Theorem 2.1). Suppose $\mathcal{H} = \{h_1, \ldots, h_n\}$, where $h_i(\lambda) = a_i^T \lambda + d_i$. If $a_i = 0$, sign(h_i) is simply the sign of d_i, and no oracle calls are needed. Thus, the presence of h_i's with $a_i = 0$ can only help. We shall henceforth assume that $a_i \neq 0$, for $i = 1, \ldots, n$. In this case, each affine function h_i corresponds to a hyperplane $H_i \in \mathbf{R}^d$, where $H_i = \{\lambda : h_i(\lambda) = 0\}$. Computing sign$(h_i)$ is thus equivalent to determining whether H_i intersects Λ and, if not, which side of H_i contains Λ. We shall find it convenient to deal interchangeably with hyperplanes and affine functions and to extend the weight function w to these hyperplanes by making $w(H_i) = w(h_i)$.

The numbers $\beta(d)$ and $\alpha(d)$ are derived recursively with respect to the dimension. For $d = 1$, the hyperplanes are n real numbers $\lambda_1, \ldots, \lambda_n$. In this case, WEIGHTED-SEARCH finds the weighted median λ_m, inquires about its position relative to λ^*, and resolves either $\{\lambda_i : \lambda_i \leq \lambda_m\}$ or $\{\lambda_i : \lambda_i \geq \lambda_m\}$. Thus, $\beta(1) = 1$ and $\alpha(1) = 1/2$. For $d \geq 2$ we proceed as follows.

Form a set $\mathcal{H}_\infty = \{H_i : a_{i2} = 0\}$. Each $H_i \in \mathcal{H} - \mathcal{H}_\infty$ intersects the λ_1–λ_2 plane (i.e., the plane where $\lambda_i = 0$ for $i \notin \{1, 2\}$) in a straight line $a_{i1}\lambda_1 + a_{i2}\lambda_2 = b_i$. Since for every scalar $l \neq 0$, sign$(h(\lambda)) =$ sign$(l) \cdot$ sign$(h(\lambda)/l)$, we can rewrite the equations of these hyperplanes so that $a_{i1} \geq 0$. Let the *slope* α_i of H_i be the same as that of $a_{i1}\lambda_1 + a_{i2}\lambda_2 = b_i$ with respect to $\lambda_2 = 0$; i.e., let $\alpha_i = (-a_{i1}/a_{i2})$. Let α^* be the weighted median of the set $\{\alpha_i\}$ where the weight of α_i is $w(H_i)$. Now we make the slopes of roughly weighted half of the hyperplanes nonnegative and weighted half nonpositive by using the change of variables $\lambda_2 = \lambda_2' + \alpha^*\lambda_1$ and $a_{i1} = a_{i1}' - \alpha^* a_{i2}$. This change of variables is only done to simplify the exposition and, indeed, needs to be reversed before making an oracle call. For convenience, we now drop the primes on λ_2' and a_{i1}'. Recalculate the slopes of the hyperplanes after making this change in variables. All hyperplanes that originally had a slope of α^* will have 0 slope. Let $\mathcal{H}_0 = \{H_i : \alpha_i = 0\}$, $\mathcal{H}_- = \{H_i : \alpha_i < 0\}$, and $\mathcal{H}_+ = \{H_i : \alpha_i > 0\}$.

Let $m = w(\mathcal{H}_\infty) + w(\mathcal{H}_0)$ and $W = w(\mathcal{H})$. Since 0 is our new weighted median slope, $w(\mathcal{H}_-) \leq (W - w(\mathcal{H}_\infty))/2 \leq W/2$ and $w(\mathcal{H}_-) + w(\mathcal{H}_0) \geq (W - w(\mathcal{H}_\infty))/2$. Therefore, $w(\mathcal{H}_-) \geq W/2 - w(\mathcal{H}_\infty)/2 - w(\mathcal{H}_0) \geq W/2 - m$. Similarly, $W/2 \geq w(\mathcal{H}_+) \geq (W/2 - m)$. Thus, sets $\mathcal{H}_-$, $\mathcal{H}_+$ and the number m satisfy the preconditions of PAIRING—assuming, without loss of generality, that $w(\mathcal{H}_-) \leq w(\mathcal{H}_+)$. WEIGHTED-SEARCH calls PAIRING$(\mathcal{H}_-, \mathcal{H}_+, m)$. Let $S_1, \ldots, S_l$, e be the sets and the element returned, where $S_i = \{c_i\} \cup D_i$. By output condition (P3) of PAIRING,

$$\sum_{i=1}^{l} w(c_i) + w(e) + m \geq W/6. \tag{1}$$

Next, we resolve the hyperplane associated with e, denoted by H_e, by calling the oracle directly. If H_e intersects Λ, we return H_e; otherwise, for the hyperplanes corresponding to elements in $S_1, \ldots, S_l$, we do the following.

Suppose that for each set $S_i = \{c_i\} \cup D_i$, c_i corresponds to a hyperplane $H_i \in \mathcal{H}_-$ and that D_i has a corresponding set of hyperplanes $\{H_{i1}, H_{i2}, \ldots, H_{iq_i}\} \subseteq \mathcal{H}_+$. (The analysis for the case where c_i is associated with a hyperplane in $\mathcal{H}_+$ is completely analogous.) For each i, form pairs $(H_i, H_{i1}), (H_i, H_{i2}), \ldots, (H_i, H_{iq_i})$. By Lemma 2.3, for each i and j,

$$w(H_{ij}) \leq w(H_i). \tag{2}$$

Consider a typical pair (H_i, H_{ij}). Since H_i and H_{ij} have strictly negative and strictly positive slopes, respectively, their intersection is a $(d-1)$-dimensional hyperplane. Through this intersection, we can draw hyperplanes $H_{ij}^{(1)}$ and $H_{ij}^{(2)}$ whose slopes are $+\infty$ and 0 respectively. Mathematically,

$$H_{ij}^{(1)}: \quad (a_{i2}a_{ij1} - a_{i1}a_{ij2})\lambda_1 = (a_{i2}b_{ij} - a_{ij2}b_i) - \sum_{r=3}^{d}(a_{i2}a_{ijr} - a_{ij2}a_{ir})\lambda_r,$$

$$H_{ij}^{(2)}: \quad (a_{i2}a_{ij1} - a_{i1}a_{ij2})\lambda_2 = (a_{ij1}b_i - a_{i1}b_{ij}) - \sum_{r=3}^{d}(a_{ij1}a_{ir} - a_{i1}a_{ijr})\lambda_r.$$

Note that $H_{ij}^{(1)}$ and $H_{ij}^{(2)}$ are $(d-1)$-dimensional hyperplanes. Now, assign a weight of $\min(w(H_i), w(H_{ij}))$ to each of $H_{ij}^{(1)}$ and $H_{ij}^{(2)}$. From equation (2), we get that $\min(w(H_i), w(H_{ij})) = w(H_{ij})$. This along with condition (P2) of PAIRING gives us

$$\sum_{r=1}^{q_i} w(H_{ir}^{(1)}) = \sum_{r=1}^{q_i} w(H_{ir}^{(2)}) = \sum_{r=1}^{q_i} w(H_{ir}) \geq w(c_i) = w(H_i). \tag{3}$$

Recursively apply WEIGHTED-SEARCH to the set of $(d-1)$-dimensional hyperplanes $\{H_{ij}^{(1)}\} \cup \mathcal{H}_\infty$. This requires $\beta(d-1)$ oracle calls. If an oracle call finds a hyperplane that intersects Λ, we return that hyperplane. Otherwise, let W_∞ and W_1 denote the weights of the hyperplanes resolved from sets $\mathcal{H}_\infty$ and $\{H_{ij}^{(1)}\}$, respectively, and let $w(\{H_{ij}^{(1)}\}) = \sum_{i=1}^{l}\sum_{r=1}^{q_i} w(H_{ir})$. Then,

$$W_\infty + W_1 \geq \alpha(d-1)\Big(w(\mathcal{H}_\infty) + w(\{H_{ij}^{(1)}\})\Big). \tag{4}$$

Let $\mathcal{H}^{(2)}$ be the set of hyperplanes in $\{H_{ij}^{(2)}\}$ for which the corresponding $H_{ij}^{(1)}$'s have been resolved in the previous step. Recursively apply WEIGHTED-SEARCH to the set of $(d-1)$-dimensional hyperplanes in $\mathcal{H}^{(2)} \cup \mathcal{H}_0$. This requires at most $\beta(d-1)$ oracle calls. As before, if we find a hyperplane which intersects Λ, we return that hyperplane; otherwise, we proceed as follows. Letting W_0 and W_{12} denote the weights of the hyperplanes resolved from sets $\mathcal{H}_0$ and $\mathcal{H}^{(2)}$, respectively, we have

$$W_0 + W_{12} \geq \alpha(d-1)(w(\mathcal{H}_0) + W_1). \tag{5}$$

To summarize the algorithm up to this point, observe that we have either found a hyperplane which intersects Λ or, from the original set $\mathcal{H}$, we have resolved an element e of weight $w(e)$, a subset of weight W_∞ of the planes in $\mathcal{H}_\infty$, and a subset of weight W_0 of the planes in $\mathcal{H}_0$. In addition to this, we have resolved a subset of weight W_1 of the hyperplanes in set $\{H_{ij}^{(1)}\}$ and a subset of weight W_{12} of the hyperplanes in set $\mathcal{H}^{(2)}$. For each hyperplane contributing to W_{12}, we have also resolved its pair in the set $\{H_{ij}^{(1)}\}$. However, W_1 and W_{12} represent the total weights of sets of auxiliary hyperplanes, rather than elements of $\mathcal{H}$. We shall now show that by resolving such auxiliary hyperplanes, we are guaranteeing the resolution of sufficiently many hyperplanes from $(\mathcal{H}_- \cup \mathcal{H}_+) - \{H_e\}$.

LEMMA 2.5. *Let W_a be the total weight of the hyperplanes resolved in $(\mathcal{H}_- \cup \mathcal{H}_+) - \{H_e\}$. Then, $W_a \geq W_{12}/2$.*

Proof. Consider a particular set $S_i = \{H_i\} \cup \{H_{i1}, H_{i2}, \ldots, H_{iq_i}\}$. The auxiliary hyperplanes formed by the intersection of hyperplanes in S_i are

$$(H_{i1}^{(1)}, H_{i1}^{(2)}), \ldots, (H_{iq_i}^{(1)}, H_{iq_i}^{(2)}).$$

Suppose $H_{i1}^{(2)}, H_{i2}^{(2)}, \ldots, H_{ip}^{(2)}$ were resolved in the second recursive call. Then these hyperplanes contributed to W_{12}. Hence, the contribution, C_i, of the auxiliary hyperplanes resulting from S_i to W_{12} is $C_i = \sum_{j=1}^{p} w(H_{ij}^{(2)})$, and W_{12} can be written as

$$W_{12} = \sum_{i=1}^{l} C_i. \tag{6}$$

For each $H_{ij}^{(2)}$ that gets resolved, its corresponding $H_{ij}^{(1)}$ has already been resolved in the first recursive call. We now rely on an observation of Megiddo [27], who noted that if we know the position of Λ relative to both $H_{ij}^{(1)}$ and $H_{ij}^{(2)}$, we can determine the position of Λ relative to at least one of H_i and H_{ij}. Let R_i be the sum of the weights of the hyperplanes resolved from S_i. Since $\cup_{i=1}^{l} S_i \subseteq (\mathcal{H}_- \cup \mathcal{H}_+) - \{H_e\}$ and the S_i's are disjoint,

$$W_a \geq \sum_{i=1}^{l} R_i. \tag{7}$$

We have two cases to consider:

Case I. In each pair, H_{ij} is resolved. Then due to equation (3), we have

$$R_i = \sum_{j=1}^{p} w(H_{ij}) = \sum_{j=1}^{p} w(H_{ij}^{(2)}) = C_i.$$

Case II. H_i is resolved in at least one pair. Then equation (3) along with condition (P2) of PAIRING implies that

$$R_i \geq w(H_i) = w(c_i) > \sum_{j=1}^{l} w(H_{ij})/2 \geq \sum_{j=1}^{p} w(H_{ij}^{(2)})/2 = C_i/2.$$

Therefore, in either case $R_i \geq C_i/2$. This along with equations (6) and (7) gives us the following:

$$W_a \geq \sum_{i=1}^{l} R_i \geq \sum_{i=1}^{l} C_i/2 = W_{12}/2.$$

Therefore, $W_a \geq W_{12}/2$. □

Let W_T be the total weight of the hyperplanes from $\mathcal{H}$ that are resolved by our algorithm; i.e., $W_T = w(e) + W_\infty + W_0 + W_a$. Using Lemmas 2.4 and 2.5 and equations (3)–(5), we have

$$\begin{aligned}
W_T &\geq w(e) + W_\infty + W_0 + W_{12}/2 \\
&\geq w(e) + W_\infty + \alpha(d-1) \cdot (w(\mathcal{H}_0) + W_1)/2 \\
&\geq w(e) + \alpha(d-1) \cdot \left(w(\mathcal{H}_0) + \alpha(d-1) \cdot (w(\mathcal{H}_\infty) + w(\{H_{ij}^{(1)}\})) \right)/2 \\
&\geq \alpha(d-1)^2 \cdot \left(w(e) + w(\mathcal{H}_0) + w(\mathcal{H}_\infty) + \sum_{i=1}^{l} w(c_i) \right)/2 \\
&\geq \alpha(d-1)^2 \cdot W/12.
\end{aligned}$$

From the preceding discussion, we conclude that the number of oracle calls satisfies $\beta(d) = 2\beta(d-1) + 1$, with $\beta(1) = 1$, and that the fraction of the total weight satisfies $\alpha(d) \geq \alpha(d-1)^2/12$, with $\alpha(1) = 1/2$. Hence, $\beta(d) = 2^d - 1$ and $\alpha(d) = 12/24^{2^{d-1}}$.

The same arguments as in [17] can be used to show that the total work done by WEIGHTED-SEARCH is $O(n)$. We omit the details.

2.3. Improving the efficiency of the search. Following Dyer [17], the *efficiency* of a search scheme is the ratio $e = \alpha(d)/\beta(d)$. As for unweighted search, the efficiency of a weighted search scheme will affect the running time of the algorithms that use the scheme as a subroutine. The search scheme we have just presented has e that is doubly exponentially small in d. Borrowing ideas from [17], we shall sketch how to make the efficiency singly exponentially small.

Let us write $\mathcal{S}(d, \beta, \alpha)$ to denote a weighted search scheme that, given a set of weighted affine functions in $\mathbf{R}^d$ of total weight W, resolves a fraction of total weight $\alpha \cdot W$ using β oracle calls. Thus, the algorithm that we have developed can be denoted by $\mathcal{S}(d, 2^d - 1, 12/24^{2^{d-1}})$. Suppose that we have an $\mathcal{S}(d-1, \beta(d-1), \alpha(d-1))$ scheme S_{d-1}. To obtain a search procedure for $\mathbf{R}^d$, proceed as follows. First, construct a $\mathcal{S}(d-1, \beta', \alpha')$ procedure with

$$\alpha' = 1 - (1 - \alpha(d-1))^r \quad \text{and} \quad \beta' = r \cdot \beta(d-1)$$

by carrying out r iterations, each of which consists of applying S_{d-1} and removing the hyperplanes that are resolved. Next, use $\mathcal{S}(d-1, \beta', \alpha')$ and the pairing scheme described earlier to obtain a procedure $\mathcal{S}(d, \beta'', \alpha'')$, where

$$\alpha'' = [1 - (1 - \alpha(d-1))^r]^2/12 \quad \text{and} \quad \beta'' = 2 \cdot r \cdot \beta(d-1) + 1.$$

Applying this procedure l times gives us a procedure $\mathcal{S}(d, \beta(d), \alpha(d))$ that solves d-dimensional hyperplanes with

$$\beta(d) = l \cdot \beta'' = l(2r \cdot \beta(d-1) + 1)$$

and

$$\alpha(d) = 1 - (1 - \alpha'')^l = 1 - \left\{ 1 - \frac{1}{12}[1 - (1 - \alpha(d-1))^r]^2 \right\}^l.$$

We can use this framework to obtain a scheme $\mathcal{S}(d, \beta(d), \alpha(d))$, where $\alpha(d) \geq 1/12$ for all d. For $d = 1$, we can easily obtain a scheme $\mathcal{S}(1, 1, 1/2)$. Suppose $\alpha(d-1) \geq 1/12$. If we choose $l = 2$ and $r = 15$, we get

$$\alpha(d) = 1 - \left\{ 1 - \frac{1}{12}[1 - (1 - 1/12)^{15}]^2 \right\}^2 \geq 1/12$$

as desired. Now

$$\beta(d) = 2 \cdot (2 \cdot 15 \cdot \beta(d-1) + 1) \leq 2(60^{d-1}),$$

and we have a procedure $\mathcal{S}(d, 2(60^{d-1}), 1/12)$ whose efficiency is singly exponentially small.

3. Convex optimization in fixed dimension. An algorithm is *piecewise affine* if the only operations it performs on intermediate values that depend on the input numbers are additions, multiplications by constants, copies, and comparisons [14], [15]. Several well-known algorithms fall into this category, including many network-optimization algorithms [13], [33]. Suppose $\mathcal{Q} \subseteq \mathbf{R}^d$ is a (possibly empty) convex set defined by a set of at most l linear inequalities, where l is some fixed integer. Let $g : \mathcal{Q} \to \mathbf{R}$ be a concave function. Our goal is to compute

$$g^* = \max\{g(\lambda) : \lambda \in \mathcal{Q}\} \tag{8}$$

or, if $\mathcal{Q} = \emptyset$, to return a message that this problem is infeasible. Cohen and Megiddo [14], [15] showed that, if g is computable by a piecewise affine algorithm that runs in time T and makes D sets of at most M comparisons, then problem (8) can be solved in $O((D \log M)^d T)$ time. Closely related results were obtained by Norton, Plotkin, and Tardos [29] and Aneja and Kabadi [4]. Toledo has extended this work to problems involving piecewise polynomial functions [34]. The main result of this section is to show that weighted multidimensional search in conjunction with Cole's circuit-simulation technique [16] can sometimes be used to solve (8) in $O((D^d + \log^d M)T)$ time.

To streamline the presentation, for the most part we shall omit any mention of constants that depend on d. The magnitude of these values is discussed in §3.3.

3.1. The basic scheme. We now review the solution scheme of Megiddo and Cohen [14], [15] and Aneja and Kabadi [4] as it forms the basis for our algorithm. Our presentation is somewhat simpler, among other reasons because it avoids the notion of "minimal weak approximation" used in [14]. We shall assume that problem (8) is bounded. This is done without loss of generality, since unbounded problems can be handled by Seidel's technique of adding "constraints at infinity" [32]. Note also that if g is computable by a piecewise affine algorithm, it is the lower envelope of a finite set of linear functions [14]. We say that a linear function $f : \mathbf{R}^d \to \mathbf{R}$ is *active* at $\lambda^{(0)} \in \mathcal{Q}$ if $g(\lambda^{(0)}) = f(\lambda^{(0)})$ and $g(\lambda) \leq f(\lambda)$ for all $\lambda \in \mathcal{Q}$ and we shall write Λ to denote the set of maximizers of g.

Let us refer to the algorithm that solves a d-dimensional problem of the form (8) as algorithm $\mathcal{C}^d$. Let H be a hyperplane in $\mathbf{R}^d$, and let g_H^* denote the maximum of g on H; i.e.,

$$g_H^* = \max\{g(\lambda) : \lambda \in H \cap \mathcal{Q}\}.$$

Suppose we have an oracle $\mathcal{B}^d$ that, as in §2, returns $\text{sign}_\Lambda(h)$ for any given affine function h. Moreover, if h defines a hyperplane H, $\mathcal{B}^d$ returns g_H^*, assuming $H \cap \mathcal{Q} \neq \emptyset$.

Obviously, $\mathcal{A}$ can play the role of $\mathcal{C}^0$. For $d \geq 1$, $\mathcal{C}^d$ proceeds as follows. First, it determines whether $\mathcal{Q}$ is empty and, if so, returns a message saying that (8) is infeasible. Since $\mathcal{Q}$ is defined by a fixed number of linear inequalities, this takes $O(1)$ time. If $\mathcal{Q} \neq \emptyset$, $\mathcal{C}^d$ uses Megiddo's algorithm simulation technique [25], [26] to do one of two things. The first option is to find a hyperplane H defined by $h(\lambda) = 0$ such that $\text{sign}_\Lambda(h) = 0$. Then, clearly, $g^* = g_H^*$. The second option is to find a linear function f and a set of linear inequalities $\mathcal{L}$ defining a nonempty convex set $\mathcal{Q}^* \subseteq \mathcal{Q}$ such that

(C1) $\mathcal{Q}^* \subseteq \Lambda$ and

(C2) f is active at every $\lambda \in \mathcal{Q}^*$.

In this case, solving (8) reduces to solving the linear programming problem

$$\max\{f(\lambda) : \lambda \in \mathcal{Q}^*\},$$

which can be done in time linear in $\mathcal{L}$, since d is fixed [27]. Algorithm $\mathcal{C}^d$ relies on the observation that, because $\mathcal{A}$ is piecewise affine and its inputs are linear functions of λ, all the intermediate values of its real variables can be represented implicitly as linear forms in λ. Using this representation, a single computation path of $\mathcal{A}$, may correspond to the evaluation of $g(\lambda)$ for a *set* of distinct λ-values.

Suppose that for $s \leq r$, we know how to find a set $\mathcal{Q}' \subset \mathbf{R}^d$ defined by a set of linear inequalities $\mathcal{L}$ such that $\mathcal{Q}' \cap \Lambda \neq \emptyset$ and such that the outcomes of the first r steps of any computation path of $\mathcal{A}$ for every $\lambda^* \in \mathcal{Q}'$ are exactly the same (when values are represented implicitly). We wish to find such a set for $s = r + 1$. Before proceeding, note that finding $\mathcal{Q}'$ when $s = 0$ is trivial, since we can choose $\mathcal{Q}' = \mathcal{Q}$. For $s = r + 1$, observe that knowing the outcomes of the first r steps tells us what the $(r + 1)$st step of $\mathcal{A}$ will be; we now need to determine the outcome of this step. If the $(r+1)$st step is an addition of two or more numbers, a multiplication by a constant, or an assignment, $\mathcal{C}^d$ does the corresponding operations with linear forms and proceeds to the next step of $\mathcal{A}$.

If the $(r+1)$st step is a comparison between two variables, $\mathcal{C}^d$ compares the corresponding linear forms $f_1(\lambda)$ and $f_2(\lambda)$ using $\mathcal{B}^d$ to resolve the function $h(\lambda) = f_1(\lambda) - f_2(\lambda)$. Suppose $h(\lambda) = 0$ defines a hyperplane H. If $\text{sign}_\Lambda(h) = 0$, then g^* is the value of g^*_H returned by the oracle, and $\mathcal{C}^d$ halts. Otherwise, $\mathcal{C}^d$ updates $\mathcal{L}$ by adding the inequality $h(\lambda) > 0$ if $\text{sign}(h) = +1$ or the inequality $h(\lambda) < 0$ if $\text{sign}(h) = -1$. The next step to be simulated from $\mathcal{A}$ will be the action corresponding to $f_1(\lambda) > f_2(\lambda)$ or $f_1(\lambda) < f_2(\lambda)$ depending on whether $\text{sign}(h)$ is $+1$ or -1. If h is a constant function, the oracle's job is trivial, since the outcome of the comparison is independent of λ and the simulation proceeds accordingly.

If $\mathcal{C}^d$ simulates $\mathcal{A}$ to completion, $\mathcal{Q}'$ will satisfy condition (C1). Furthermore, the output of $\mathcal{A}$ will be a linear function f satisfying condition (C2). Since $\mathcal{A}$ does $O(T)$ comparisons, $|\mathcal{L}| = O(T)$ and the resulting linear program in d variables can be solved in $O(T)$ time [27]. The total time for algorithm $\mathcal{C}^d$ is therefore $O(T \cdot b(d))$, where $b(d)$ is the running time of $\mathcal{B}^d$. We now turn our attention to the implementation of $\mathcal{B}^d$.

Implementing the oracle. Let $h(\lambda) = \sum_{i=1}^{d} a_i\lambda_i + b$ be the function to be resolved. If $a_i = 0$ for $i = 1, \ldots, n$, $\text{sign}(h)$ depends simply on the sign of b. Otherwise, $H = \{\lambda : h(\lambda) = 0\}$ is a hyperplane in $\mathbf{R}^d$. To resolve h, $\mathcal{B}^d$ first determines if $H \cap \mathcal{Q} = \emptyset$. If this is so, then, since $\Lambda \subseteq \mathcal{Q}$, we simply need to find a point $\lambda^{(0)} \in \mathcal{Q}$ and evaluate $\text{sign}_{\{\lambda^{(0)}\}}(h)$. Determining if the intersection of H and $\mathcal{Q}$ is nonempty and finding a point inside $\mathcal{Q}$ take $O(1)$ time, since $\mathcal{Q}$ is defined by a set of $O(l)$ linear inequalities and l is fixed.

From now on, assume $H \cap \mathcal{Q} \neq \emptyset$. Now, if $\Lambda \cap H = \emptyset$, due to concavity of g, the set of all points λ' such that $g(\lambda') > g^*_H$ is contained in only one side of H. This observation leads to the following result, which is the basis for the implementation of $\mathcal{B}^d$.

LEMMA 3.1. *Let $H = \{\lambda : h(\lambda) = 0\}$ be a hyperplane in $\mathbf{R}^d$, and for every real number a, let $H(a)$ denote the hyperplane given by $H(a) = \{\lambda : h(\lambda) = a\}$. Then,*

$$sign_\Lambda(h) = \begin{cases} +1 & \textit{if } (\exists\epsilon > 0)[g^*_{H(\epsilon)} > g^*_H], \\ -1 & \textit{if } (\exists\epsilon > 0)[g^*_{H(-\epsilon)} > g^*_H], \\ 0 & \textit{otherwise}. \end{cases}$$

Furthermore, if $sign_\Lambda(h) = +1$ $[sign_\Lambda(h) = -1]$, then $g^*_{H(\gamma)} > g^*_H$ $[g^*_{H(-\gamma)} > g^*_H]$ for all $\gamma \in (0, \epsilon]$, where $\epsilon > 0$ is sufficiently small.

The lemma tacitly assumes that $H(\epsilon) \cap \mathcal{Q} \neq \emptyset$ $[H(-\epsilon) \cap \mathcal{Q} \neq \emptyset]$ for some $\epsilon > 0$. If $H(\epsilon) \cap \mathcal{Q} = \emptyset$ $[H(-\epsilon) \cap \mathcal{Q} = \emptyset]$ for every $\epsilon > 0$, we can immediately conclude that

$\text{sign}(h) \neq +1$ [$\text{sign}(h) \neq -1$]. Thus, we shall continue assuming that $H(\epsilon) \cap \mathcal{Q} \neq \emptyset$ and $H(-\epsilon) \cap \mathcal{Q} \neq \emptyset$ for some $\epsilon > 0$.

Lemma 3.1 implies that we can implement $\mathcal{B}^d$ by computing g_H^*, $g_{H(\epsilon)}^*$ and $g_{H(-\epsilon)}^*$ for sufficiently small $\epsilon > 0$. Computing g_H^* is a $(d-1)$-dimensional problem of the same form as that of computing g^*; hence, g_H^* can be calculated by recursively calling $\mathcal{C}^{d-1}$. We can also compute $g_{H(\epsilon)}^*$ using $\mathcal{C}^{d-1}$, provided $\mathcal{C}^{d-1}$ treats ϵ as a symbolic constant whose only known attribute is being arbitrarily small and positive. (The details of computing $g_{H(-\epsilon)}^*$ are analogous and therefore omitted.) The output $g_{H(\epsilon)}^*$ of this execution of $\mathcal{C}^{d-1}$ will depend linearly on ϵ; i.e., $g_{H(\epsilon)}^* = g_0 + g_1\epsilon$. The values of $g_{H(\epsilon)}^*$ and g_H^* are compared by computing $y = \text{sign}(g_H^* - g_{H(\epsilon)}^*) = \text{sign}((g_H^* - g_0) - g_1\epsilon)$. If $|g_H^* - g_0| > 0, d = \text{sign}(g_H^* - g_0)$, since ϵ is arbitrarily small. Otherwise, $d = \text{sign}(-g_1)$, since ϵ is positive. Of course, $\mathcal{C}^{d-1}$ will itself call $\mathcal{B}^{d-1}$, which will introduce a perturbation of its own. In order to deal effectively with the various symbolic perturbations, we shall establish a certain ordering among them.

The state of the execution of $\mathcal{C}^d$ is partially described by sequence of currently active procedure calls (i.e., calls that have not yet been completed). Let us follow one sequence of procedure calls $\mathcal{C}^d \to \mathcal{B}^d \to \mathcal{C}^{d-1} \to \mathcal{B}^{d-1} \to \cdots \to \mathcal{B}^{d-r+1} \to \mathcal{C}^{d-r}$. Within this sequence, for $0 \leq j \leq r-1$, $\mathcal{B}^{d-j} \to \mathcal{C}^{d-j-1}$ corresponds to one of the three calls to $\mathcal{C}^{d-j-1}$ done by $\mathcal{B}^{d-j}$; we refer to this part of the sequence as *level* j. Each level reduces the dimension of the problem by one. Also, depending on which of the three calls the level corresponds to, the call may or may not introduce a perturbation. If it does, we shall refer to the perturbation as ϵ_j. Let $\mathcal{J} = \{i_1, \ldots, i_s\} \subseteq \mathcal{I} = \{0, \ldots, r\}$ consist of all j's such that a perturbation is introduced up to level j. We assume that $0 \leq i_1 \leq r$ and, for $0 \leq j \leq s-1$, $0 \leq i_j < i_{j+1} \leq r$. The set $\mathcal{J}$ indicates which perturbations are "active" at the current stage of the execution of $\mathcal{C}^d$. The problem to be solved at level r can thus be expressed as

$$g_r^* = \max\{g(\lambda) : \lambda \in \mathcal{Q}'(\epsilon_{i_1}, \ldots, \epsilon_{i_s})\},$$

where $\mathcal{Q}'(\epsilon_{i_1}, \ldots, \epsilon_{i_s})$ is a $(d-r)$-dimensional subset of $\mathbf{R}^d$ defined by the intersection of $O(d)$ linear inequalities in $\{\lambda_i\}$ and $\epsilon_{i_1}, \ldots, \epsilon_{i_s}$.

Now, suppose $\mathcal{C}^{d-r}$ invokes $\mathcal{B}^{d-r}$ to resolve a hyperplane

$$H(\epsilon_{i_1}, \ldots, \epsilon_{i_s}) = \{\lambda : h(\lambda, \epsilon_{i_1}, \ldots, \epsilon_{i_s}) = 0\}.$$

For every real number a, let $H'(\epsilon_{i_1}, \ldots, \epsilon_{i_s}, a) = \{\lambda : h(\lambda, \epsilon_{i_1}, \ldots, \epsilon_{i_s}) = a\}$. Then, applying Lemma 3.1, $\mathcal{B}^d$ solves three problems:

$$\begin{aligned}
g_{r+1}^*(0) &= \max\{g(\lambda) : \lambda \in \mathcal{Q}'(\epsilon_{i_1}, \ldots, \epsilon_{i_s}) \cap H'(\epsilon_{i_1}, \ldots, \epsilon_{i_s}, 0)\},\\
g_{r+1}^*(\epsilon_{r+1}) &= \max\{g(\lambda) : \lambda \in \mathcal{Q}'(\epsilon_{i_1}, \ldots, \epsilon_{i_s}) \cap H'(\epsilon_{i_1}, \ldots, \epsilon_{i_s}, \epsilon_{r+1})\},\\
g_{r+1}^*(-\epsilon_{r+1}) &= \max\{g(\lambda) : \lambda \in \mathcal{Q}'(\epsilon_{i_1}, \ldots, \epsilon_{i_s}) \cap H'(\epsilon_{i_1}, \ldots, \epsilon_{i_s}, -\epsilon_{r+1})\},
\end{aligned}$$

where $\epsilon_{r+1} > 0$. By Lemma 3.1, if there exists an $\epsilon_{r+1} > 0$ such that $g_{r+1}^*(\epsilon_{r+1}) > g_{r+1}^*(0)$, then $g_{r+1}^*(\gamma) > g_{r+1}^*$ for *any* $\gamma \in (0, \epsilon_{r+1}]$. Thus, when dealing symbolically with ϵ_{r+1}, we can assume that it is arbitrarily smaller than any one of $\epsilon_{i_1}, \ldots, \epsilon_{i_s}$. By the same reasoning, when dealing with perturbations $\epsilon_{i_1}, \ldots, \epsilon_{i_s}$, we can always act under the assumption that

$$0 < \epsilon_{i_s} \ll \epsilon_{i_{s-1}} \ll \cdots \ll \epsilon_{i_2} \ll \epsilon_{i_1} \ll 1. \tag{9}$$

Since $\mathcal{A}$ is piecewise affine, all numbers manipulated at any level of the execution of $\mathcal{C}^d$ are linear forms in the λ_i's and the ϵ_i's. Suppose the execution of $\mathcal{C}^d$ produces a sequence of procedure calls that eventually triggers a comparison between two values. If the values involve λ (and, possibly, some ϵ_i's), the comparison will be handled by an oracle call. Otherwise, we will be comparing linear forms in the $\epsilon_i's$. For a correct implementation of $\mathcal{C}^d$, it suffices to deal properly with the second kind of comparison. Suppose the two numbers have the form $u = u_0 + \sum_{j=1}^{s} u_j \epsilon_{i_j}$ and $v = v_0 + \sum_{j=1}^{s} v_j \epsilon_{i_j}$. We must compute sign$(t)$, where $t = t_0 + \sum_{j=1}^{s} t_j \epsilon_{i_j}$ and $t_j = u_j - v_j$, $j = 1, \dots, s$. Obviously, if $t_j = 0$ for $j = 0, 1, \dots, s$, sign$(t) = 0$. Otherwise, there is a smallest subscript d, $0 \leq d \leq s$, such that $|t_d| > 0$. By (9), $\epsilon_{i_{d+1}}, \dots, \epsilon_{i_s}$ can be assumed to be arbitrarily smaller than ϵ_{i_d}. Thus sign$(t) =$ sign(t_d). We should note that the use of perturbation techniques is common in mathematical programming [13], [31], one example being the *lexicographic rule* applied in the simplex algorithm. These methods have also found applications in computational geometry [18]. An earlier application of perturbation methods to parametric computing was given by Megiddo [28].

Let $c(d)$ be the running time of $\mathcal{C}^d$. Since at any level, the number of perturbations that $\mathcal{C}^j$ will have to deal with is d, and d is fixed, the running time of $\mathcal{C}^j$ will be the same, asymptotically, whether it deals with a perturbed or an unperturbed problem. As we have seen, $c(d)$ is $O(T \cdot b(d))$, where $b(d)$ is the running time of $\mathcal{B}^d$, and $b(d)$ is $O(c(d-1))$ because $\mathcal{B}^d$ is implemented via three recursive calls to $\mathcal{C}^{d-1}$. Since $c(0) = O(T)$, we conclude that $c(d)$ is $O(T^{d+1})$.

3.2. Speeding up the search. The main bottleneck in algorithm $\mathcal{C}^d$ is the need to apply oracle $\mathcal{B}^d$ to each affine function generated during the simulation of algorithm $\mathcal{A}$. One way to reduce this problem is to arrange things so that by using a small number of oracle calls, we are able to resolve a large number of functions. Megiddo [26] proposed a way to do this in the context of one-dimensional problems, an idea that has subsequently been used in multidimensional optimization [14], [29]. Megiddo's approach is to simulate the execution of a *parallel* algorithm $\mathcal{A}$ for computing $g(\lambda)$ rather than a sequential one. Suppose $\mathcal{A}$ uses M processors and carries out at most D parallel steps. In each step of the simulation, a *batch* of at most M comparisons is carried out. In $\mathcal{C}^d$'s simulation of $\mathcal{A}$, each such comparison has an associated affine function h which can be resolved using $\mathcal{B}^d$. Every parallel step produces a set of $O(M)$ hyperplanes. By using Theorem 2.1, we either resolve these $O(M)$ hyperplanes with $O(\log M)$ oracle calls or find a hyperplane H which intersects Λ. In the latter case, we reduce the original problem to the $(d-1)$-dimensional problem of computing the maximum g_H^* on $H \cap \Lambda$, because this maximum will also be a global maximum. Since $\mathcal{B}^d$ is implemented by making at most three recursive calls to $\mathcal{C}^{d-1}$, the running time of $\mathcal{C}^d$ is $O(c(d-1) \cdot D \cdot \log M + T)$, where $c(d)$ denotes the running time of $\mathcal{C}^d$. Since $c(0) = O(T)$, the running time of $\mathcal{C}^d$ will be $O((D \log M)^d T)$.

Cole [16] showed that one can improve on Megiddo's results for certain important special cases. Like Megiddo's method, Cole's technique applies to one-dimensional parametric search problems, but we shall show that it can be extended to higher dimensions. What follows shall require some elementary knowledge of *combinational circuits* as described, say, in [12]. A *combinational circuit* $\mathcal{G}$ is a directed acyclic graph whose nodes are *combinational elements* (e.g., adders, min gates, etc.) and where an edge from element e_1 to element e_2 implies that the output of e_1 is an input to e_2. Elements of zero fan-in are *inputs*; elements of zero fan-out are *outputs*. An element is said to be *active* if all its inputs are known, but the associated operation has not been carried out yet. An element is said to have been *resolved* when the associated operation has been carried out.

Now, suppose that the algorithm $\mathcal{A}$ simulated by $\mathcal{C}^d$ is implemented as a combinational circuit $\mathcal{G}$ (which is given to us explicitly) of width M and depth D, whose elements are multiplier gates where one of the two inputs is a constant, min gates, adders, and subtractors. Megiddo's approach would simulate $\mathcal{G}$ level by level, in D steps, where each step would carry out the operations of the gates at a given level. The operations within a level would be carried out using Theorem 2.1, with $O(\log M)$) calls to the oracle $\mathcal{B}^d$. In Cole's approach, each step only resolves a fixed fraction of of the active nodes, using only a constant number of oracle calls. The choice of which nodes to resolve is guided by a weight function $w : V(\mathcal{G}) \to \mathbf{R}$. To describe the strategy precisely, we will need some notation. The *active weight*, W, of the circuit is the sum of the weights of its active elements. Let $\alpha \leq 1/2$ be a positive number. An *α-oracle with respect to w* — or simply an α-oracle — is a procedure that is guaranteed to resolve a set of active elements whose total weight is at least $\alpha W/2$. The following is a restatement and an extension of a result in [16].

LEMMA 3.2. *Let $\mathcal{G}$ be a combinatorial circuit of width M and depth D. Let $d_{\min} = \min\{d_I, d_O\}$, where d_I (d_O) denotes the maximum fan-in (fan-out) of an element of $\mathcal{G}$. Then, there exists a weight function w such that $\mathcal{G}$ can be evaluated with $O(D \log d_{\min} + \log M)$ calls to an α-oracle with respect to w.*

Proof. Let the weight function w be defined as follows. The weight of each output element is 1, and the weight of each internal element is twice the sum of weights of its immediate descendants. Then scale the weights to make the total weight of input elements equal to M.

LEMMA 3.3. *At the start of the $(k+1)$st iteration, $k \geq 0$, the active weight is at most $(1-\alpha/2)^k \cdot M$.*

Proof. By induction on k. The result holds for $k = 0$ since, at the start of the first iteration, only the input elements are active and their total weight is M. To prove the inductive step, it suffices to show that at each iteration the active weight is reduced by a factor of at least $\alpha/2$.

Suppose element e is resolved. Then e ceases to be active, but all its descendants may become active. Hence, the resolution of e reduces the active weight by at least $w(e)/2$. Let the active weight of the network be W. In one step, the α-oracle resolves a set of elements whose total weight is at least $\alpha \cdot W$. Thus, in one step, the active weight is reduced from W to $(1-\alpha/2)W$. □

LEMMA 3.4. *The weight of any circuit element is at least $(2d_{\min})^{-D}$.*

Proof. After the initial weight assignment, but prior to scaling, the total weight of the elements at depth j is at most $M(2d_{\min})^{D-j}$. Thus, the total weight of the input nodes is at most $M(2d_{\min})^D$. Hence the scaling factor is at most $(2d_{\min})^D$. Since, prior to scaling, every element has a weight of at least 1, after scaling the weight of any circuit element will be at least $(2d_{\min})^{-D}$. □

LEMMA 3.5. *Let $\gamma = c(D \log 2d_{\min} + \log M)$, where $c = \lfloor 1 - 1/\log_2(1-\alpha/2) \rfloor$. Then, there will be no active elements after $k \geq \gamma$ iterations.*

Proof. First, observe that

$$\begin{aligned} \gamma &> -(D \log 2d_{\min} + \log M) / \log_2(1-\alpha/2) \\ &= -\left(\log M \cdot (2d_{\min})^D\right) / \log_2(1-\alpha/2) \\ &= \log_{(1-\alpha/2)} (M \cdot (2d_{\min})^D)^{-1}. \end{aligned} \tag{10}$$

By Lemma 3.3, the active weight at the start of the $(k+1)$st iteration is at most $(1-\alpha/2)^k \cdot M$. Using (10) and the fact that $(1-\alpha/2)$ is less than 1, we have

$$\begin{aligned} (1-\alpha/2)^k \cdot M &\leq (1-\alpha/2)^\gamma \cdot M \\ &< (1-\alpha/2)^{\log_{(1-\alpha/2)}(M \cdot (2d_{\min})^D)^{-1}} \cdot M \\ &= (2d_{\min})^{-D}. \end{aligned}$$

As the weight of any element in the circuit is at least $(2d_{\min})^{-D}$ and the active weight is strictly less than this weight, there are no active elements after the kth iteration. □

By Lemma 3.5, there will be no active elements after $c(D \log 2d_{\min} + \log M)$ steps, where c depends only on α. Thus, $\mathcal{G}$ can be evaluated with $O(D \log d_{\min} + \log M)$ calls to an α-oracle. □

In order to use Lemma 3.2, we need to give an efficient implementation of the α-oracle. We will actually implement a slight variant of the α-oracle, which will allow for early termination of the simulation in case the optimum if found at some intermediate step. Let A be the set of active elements of $\mathcal{G}$ and let $A_1 \subseteq A$ be the set of adders, subtractors, and constant multipliers. Each $e \in T_1$ can be resolved immediately by simply doing the corresponding operation on the input linear forms. The remaining active elements are comparators, every one of which has an associated affine function. Let the set of all such functions be $\mathcal{H} = \{h_1, \ldots, h_n\}$, where h_i is the function associated with $e_i \in A - A_1$, and assign a weight of $w(e_i)$ to h_i. We either resolve a fixed fraction of the functions with $O(1)$ oracle calls using algorithm WEIGHTED-SEARCH of §2 or, if at any point during its execution, WEIGHTED-SEARCH encounters a hyperplane H (even an auxiliary one) such that $H \cap \Lambda \neq \emptyset$, we return g^*_H. In either case, the running time of the α-oracle is $O(c(d-1))$, since each oracle call requires $O(1)$ calls to $\mathcal{C}^{d-1}$. Thus, Lemma 3.2 leads to an implementation of $\mathcal{C}^d$ whose running time is $O(c(d-1)(D + \log M) + D \cdot M)$. Since $c(0) = O(T)$, we can deduce that the running time of $\mathcal{C}^d$ is $O((D^d + \log^d M)T)$ (note that the weight function required for the application of Lemma 3.2 can be computed within this time bound).

3.3. Some remarks on constant factors. The use of schemes involving multidimensional search seems to lead invariably to large constants that depend on d [17]. Using standard techniques [10], [17], it can be shown that the algorithms described in this section have hidden constants of the form $2^{O(d^2)}$, provided the search algorithm with singly exponentially small efficiency is used. Some improvements are possible. For the case where all the weights are powers of 1/4, as would occur if the circuit to be simulated is a comparator-based sorter, we can obtain a search scheme with $\alpha(d) = 1/3$ and $\beta(d) = 2(20^{d-1})$; the details are technical and hence omitted. Using this improved scheme, the running time of the optimization algorithm will still have a constant of the form $2^{O(d^2)}$, but the constant inside the O will be smaller.

4. Solving the Lagrangian dual when the number of constraints is fixed. The method of Lagrangian relaxation, originally developed by Held and Karp [21], [22], is motivated by the observation that many combinatorial problems that are known to be NP-hard can be viewed as easy problems complicated by a relatively small set of side constraints. More formally, we consider optimization problems of the following sort:

$$Z_P = \min\{c^T x : Ax \leq 0, x \in X\}, \tag{11}$$

where c is a $n \times 1$ vector, A is a $d \times n$ matrix, x is a $n \times 1$ vector, and X is a polyhedral subset of $\mathbf{R}^d$. The set of inequalities $Ax \leq 0$ constitutes the complicating set of constraints in the sense that, in its absence, the problem is polynomially solvable.

The *Lagrangian relaxation* of (11) is obtained by pricing out the constraints $Ax \leq 0$ into the objective function by introducing a vector $\lambda = (\lambda_1, \ldots, \lambda_d)$ of Lagrange multipliers as follows:

$$Z_D(\lambda) = \min\{c^T x + \lambda^T Ax : x \in X\}. \tag{12}$$

It is well known that $Z_D(\lambda) \leq Z_P$ for all $\lambda \geq 0$ [20]. Thus, if there is a polynomial-time algorithm to compute $Z_D(\lambda)$ for any fixed $\lambda \geq 0$, problem (12) will provide an efficient way to obtain a lower bound on the solution to (11). Such a bound can be of great utility in branch-and-bound methods. The best lower bound on Z_P attainable via (12) is given by

$$Z^*_D = \max\{Z_D(\lambda) : \lambda \geq 0\}. \tag{13}$$

Problem (13) is the *Lagrangian dual* of (11) with respect to the set of constraints $Ax \leq 0$, and Z_D^* is the *value* of the Lagrangian dual.

Computational experiments have repeatedly shown that Z_D^* provides excellent lower bounds on the optimum solution of Z_P [20], thus motivating the search for efficient algorithms to solve the Lagrangian dual. One widely used method is *subgradient optimization*, first proposed in [22]. Despite its success in practice, this technique is not known to be a polynomial-time algorithm even if (12) can be solved in polynomial time.

It is well known that if $Z_D(\lambda)$ can be computed in polynomial time for each fixed $\lambda \geq 0$, then the Lagrangian dual can be solved in polynomial time [31]. Recently, Bertsimas and Orlin [6] have presented faster polynomial-time algorithms for certain special cases. An issue that has received some attention [4] is whether there exist *strongly polynomial* algorithms to solve the Lagrangian dual. (An algorithm is said to be strongly polynomial if the number of arithmetic operations it carries out is polynomially bounded independently of the magnitudes of the input numbers.) The algorithms discussed above are not strongly polynomial even if $Z_D(\lambda)$ can be computed in strongly polynomial time.

We shall be interested here only in the case where the number d of complicating constraints is fixed. Since Z_D is a concave function [31], if $Z_D(\lambda)$ is computable in strongly polynomial time by a piecewise affine algorithm, the results of Megiddo and Cohen described in §3 imply the existence of strongly polynomial-time algorithms to solve the Lagrangian dual. We focus our attention on two broad families of problems where weighted multidimensional search allows us to obtain faster algorithms than the Megiddo–Cohen approach: matroidal knapsack problems and a class of constrained optimum subgraph problems on graphs of bounded treewidth.

4.1. Matroidal knapsack problems. What follows presupposes some familiarity with matroid theory (see, e.g., [23]). Consider a matroid $\mathcal{M} = (E, \mathcal{G})$ where E, the *ground set*, is a finite set and $\mathcal{G}$ is a collection of certain subsets of E called *independent sets*. We assume that $\mathcal{G}$ is given in a *concise* form; i.e., there is an algorithm with running time $c(n)$, polynomial in $n = |E|$, for finding whether a given subset of E is independent. Suppose each element $e \in E$ has a *value* $v(e)$. In ordinary matroid optimization problems, one must find an optimum *base* (maximal independent set) of maximum total value. The standard algorithm for doing so is the *greedy method*, which first sorts the elements according to value and then considers the elements in nonincreasing order. An element e is added to the current set A if $A \cup \{e\}$ is independent. The greedy algorithm takes time $O(n \log n + nc(n))$.

In *multiconstrained matroidal knapsack* (MMK) problems, in addition to a value, each $e \in E$ has a d-dimensional *size* vector $s(e)$, and there is a d-dimensional *capacity* vector C. The problem is to find a base G^* such that

$$Z^* = \sum_{e \in G^*} v(e) = \max_{G \in \mathcal{G}} \left\{ \sum_{e \in G} v(e) : \sum_{e \in G} s(e) \leq C \right\}.$$

We refer the reader to Camerini et al. [11] for a discussion of the various applications of these problems, as well as for references. MMK problems are in general NP-hard. We can bound Z^* by solving its Lagrangian dual. Let

$$Z_D(\lambda) = \max_{G \in \mathcal{G}} \left\{ \sum_{e \in G} v(e) - \lambda^T \left(\sum_{e \in G} s(e) - C \right) \right\}.$$

The Lagrangian dual is

$$Z_D^* = \min\{Z_D(\lambda) : \lambda \geq 0\}. \tag{14}$$

In [11], Camerini et al. outline an algorithm for (14) whose running time is not guaranteed to be polynomial. Noting that the crucial first stage of the greedy method (where all comparisons are done) can be carried out in parallel using an $O(\log n)$-depth, $O(n)$-width sorting circuit [2], we can use the Cohen–Megiddo technique to obtain an $O((n\log n+n\cdot c(n))\cdot\log^{2d} n)$ algorithm using the approach outlined in §3, with the greedy algorithm playing the role of algorithm $\mathcal{A}$. Using Lemma 3.2, and the weighted multidimensional search algorithm, we obtain a $O((n\log n + n\cdot c(n))\cdot \log^d n)$ algorithm. We note that if the underlying matroidal problem has a more specialized structure (e.g., if it is the spanning tree problem), even faster algorithms are possible.

4.2. Constrained optimum subgraph problems. Optimum subgraph problems have the following form. Given a graph G with real-valued vertex and edge weight functions $w_V : V(G) \to \mathbf{R}$ and $w_E : E(G) \to \mathbf{R}$, respectively, find an optimum (i.e., minimum- or maximum-weight) subgraph H satisfying a property P. Well-known examples of such problems are minimum-weight dominating set, minimum-weight vertex cover, and the traveling salesman problem. Let us write $\mathsf{val}_G(H)$ to denote $\sum_{v\in V(H)} w_V(v) + \sum_{e\in E(H)} w_E(e)$, where H is a subgraph of G. We can express all optimum subgraph problems as

$$(15)\qquad z_G^P = \operatorname{opt}\{\mathsf{val}_G(H) : H \text{ a subgraph of } G \text{ satisfying } P\},$$

where "opt" is either "min" or "max," depending on the problem.

Even though many optimum subgraph problems are known to be NP-complete, several researchers have developed methodologies for devising linear-time algorithms for graphs of *bounded tree-width* [3], [5], [8], [9], [7], [35] (for a definition of tree-width, see [30]). While their approaches differ from each other in several respects, in essence they all deal with subgraph problems that have certain "regularity" properties that make them amenable to dynamic programming solutions. The class of regular problems is broad, and includes the subgraph problems mentioned above, as well as many others, such as the maximum cut problem and the Steiner tree problem (see, e.g., [3], [9], [7]).

Suppose that, in addition to a weight function, every $v \in V(G)$ ($e \in E(G)$) has a d-dimensional size vector $s_V(v)$ ($s_E(e)$). The problem is to solve (15) subject to the knapsack-like constraint

$$\sum_{v\in V(H)} s_V(v) + \sum_{e\in E(H)} s_E(e) \le t,$$

where t is a d-dimensional capacity vector. Even if the unconstrained problem is solvable in polynomial time, the constrained one may be NP-hard. Such is the case, for example, for the dominating set problem on trees (which are graphs of tree-width 1) even if $d = 1$ [24].

For every $v \in V(G)$, let $W_V(v,\lambda) = w_V(v) + \lambda^T s_V(v)$ and for every $e \in E(G)$, let $W_E(e,\lambda) = w_E(e) + \lambda^T s_E(e)$. Let us write $\mathsf{Val}_G(H,\lambda)$ to denote $\sum_{v\in V(H)} W_V(v,\lambda) + \sum_{e\in E(H)} W_E(e,\lambda)$, where H is a subgraph of G. The Lagrangian relaxation of problem (15) is

$$(16)\qquad Z_G^P(\lambda) = \operatorname{opt}\{\mathsf{Val}_G(H,\lambda) : H \text{ a subgraph of } G \text{ satisfying } P\}.$$

If property P is regular, there exists an $O(n)$-time algorithm to compute $Z_G^P(\lambda)$ for any fixed λ. Also, as proved in [19], there exists an $O(n)$-size, $O(\log n)$-depth combinational circuit that computes $Z_G^P(\lambda)$ for any fixed λ. Thus, the results of Cohen and Megiddo summarized in §3 imply that the Lagrangian dual can be solved in $O(n\log^{2d} n)$ time. Using weighted multidimensional search and Lemma 3.2, we can improve this to $O(n\log^d n)$.

Acknowledgment. We thank the referee for several useful comments.

REFERENCES

[1] R. Agarwala and D. Fernández-Baca, *Solving the Lagrangian dual when the number of constraints is fixed*, in Proc. 13th Conference on Software Technology and Theoretical Computer Science, Lecture Notes in Comput. Sci., 652 (1992), pp. 164–175.

[2] M. Ajtai, J. Komlós, and E. Szemerédi, *A $O(n \log n)$ sorting network*, in Proc. 15th Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1983, pp. 1–9.

[3] S. Arnborg, J. Lagergren, and D. Seese, *Easy problems for tree-decomposable graphs*, J. Algorithms, 12 (1991), pp. 308–340.

[4] Y. P. Aneja and S. N. Kabadi, *Polynomial algorithms for lagrangean relaxations in combinatorial problems*, manuscript.

[5] S. Arnborg and A. Proskurowski, *Linear time algorithms for NP-hard problems restricted to partial k-trees*, Discrete Appl. Math., 23 (1989), pp. 11–24.

[6] D. Bertsimas and J. B. Orlin, *A technique for speeding up the solution of the Lagrangean dual*, Math. Programming, 63 (1994), pp. 23–45.

[7] 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.

[8] H. L. Bodlaender, *Dynamic programming on graphs with bounded tree-width*, Tech. report RUU-CS-88-4, University of Utrecht, Utrecht, The Netherlands, 1988.

[9] 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.

[10] K. L. Clarkson, *Linear programming in $O(n \times 3^{d^2})$ time*, Inform. Process. Lett., 22 (1986), pp. 21–24.

[11] P. M. Camerini, F. Maffioli, and C. Vercellis, *Multi-constrained matroidal knapsack problems*, Math. Programming, 45 (1989), pp. 211–231.

[12] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, MIT Press, Cambridge, MA, 1990.

[13] V. Chvátal, *Linear Programming*, W. H. Freeman, San Francisco, 1983.

[14] E. Cohen, *Combinatorial algorithms for optimization problems*, Tech report STAN-CS-91-1366, Department of Computer Science, Stanford University, Stanford, CA, 1991.

[15] E. Cohen and N. Megiddo, *Maximizing concave functions in fixed dimension*, Complexity in Numerical Optimization, P. M. Pardalos, ed., World Scientific, Singapore, 1993, pp. 74–87.

[16] R. Cole, *Slowing down sorting networks to obtain faster sorting algorithms*, J. Assoc. Comput. Mach., 34 (1987), pp. 200–208.

[17] M. E. Dyer, *On a multidimensional search technique and its application to the Euclidean one-centre problem*, SIAM J. Comput., 15 (1986), pp. 725–738.

[18] H. Edelsbrunner, *Algorithms in Combinatorial Geometry*, Springer-Verlag, Heidelberg, 1987.

[19] D. Fernández-Baca and G. Slutzki, *Parametric problems on graphs of bounded tree-width*, J. Algorithms, 16 (1994), pp. 108–430.

[20] M. L. Fisher, *The Lagrangian relaxation method for solving integer programming problems*, Management Science, 27 (1981), pp. 1–18.

[21] M. Held and R. M. Karp, *The traveling salesman problem and minimum spanning trees*, Oper. Res., 18 (1970), pp. 1138–1162.

[22] ———, *The traveling salesman problem and minimum spanning trees: Part* II, Math. Programming, 6 (1971), pp. 6–25.

[23] E. Lawler, *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart, and Winston, New York, 1976.

[24] J. McHugh and Y. Perl, *Best location of service centers in a treelike network under budget constraints*, Discrete Math., 86 (1990), pp. 199–214.

[25] N. Megiddo, *Combinatorial optimization with rational objective functions*, Math. Oper. Res., 4 (1979), pp. 414–424.

[26] ———, *Applying parallel computation algorithms in the design of serial algorithms*, J. Assoc. Comput. Mach., 30 (1983), pp. 852–865.

[27] ———, *Linear programming in linear time when the dimension is fixed*, J. Assoc. Comput. Mach., 31 (1984), pp. 114–127.

[28] ———, *A note on sensitivity analysis in algebraic algorithms*, Tech. report RJ 4958, IBM Almaden Research Center, San Jose, CA, 1985.

[29] C. H. Norton, S. A. Plotkin, and É. Tardos, *Using separation algorithms in fixed dimension*, J. Algorithms, 13 (1992), pp. 79–98.

[30] N. Robertson and P. D. Seymour, *Graph minors* II: *Algorithmic aspects of tree-width*, J. Algorithms, 7 (1986), pp. 309–322.

[31] A. Schrijver, *Theory of Linear and Integer Programming*, Wiley, Chichester, UK, 1986.

[32] R. Seidel, *Small-dimensional linear programming and convex hulls made easy*, Discrete Comput. Geom., 6 (1991), pp. 423–434.

[33] R. E. Tarjan, *Data Structures and Network Algorithms*, Society for Industrial and Applied Mathematics, Philadelphia, 1983.

[34] S. Toledo, *Maximizing non-linear concave functions in fixed dimension*, Complexity in Numerical Optimization, P. M. Pardalos, ed., World Scientifice, Singapore, 1993, pp. 429–446.

[35] T. V. Wimer, *Linear algorithms on k-terminal graphs*, Ph.D. thesis, Tech. report URI-030, Department of Computer Science, Clemson University, Clemson, SC, 1987.

SIAM J. COMPUT.
Vol. 25, No. 1, pp. 100–116, February 1996

RAY SHOOTING AMIDST CONVEX POLYHEDRA AND POLYHEDRAL TERRAINS IN THREE DIMENSIONS*

PANKAJ K. AGARWAL† AND MICHA SHARIR‡

Abstract. We consider the problem of ray shooting in a three-dimensional scene consisting of m (possibly intersecting) convex polyhedra or polyhedral terrains with a total of n faces, i.e., we want to preprocess them into a data structure, so that the first intersection point of a query ray and the given polyhedra can be determined quickly. We present a technique that requires $O((mn)^{2+\varepsilon})$ preprocessing time and storage, and can answer ray-shooting queries in $O(\log^2 n)$ time. This is a significant improvement over previously known techniques (which require $O(n^{4+\varepsilon})$ space and preprocessing) if m is much smaller than n, which is often the case in practice. Next, we present a variant of the technique that requires $O(n^{1+\varepsilon})$ space and preprocessing, and answers queries in time $O(m^{1/4}n^{1/2+\varepsilon})$, again a significant improvement over previous techniques when $m \ll n$.

Key words. arrangements, ray shooting, range searching, data structures, parametric search, random sampling

AMS subject classifications. 52B11, 68P05, 68Q20, 68Q25

1. Introduction.

The *ray-shooting* problem can be defined as follows:

> *Given a collection Γ of n objects in $\mathbb{R}^d$, preprocess Γ into a data structure so that one can quickly determine the first object of Γ intersected by a query ray.*

The ray-shooting problem has received much attention in the last few years because of its applications in computer graphics and other geometric problems [1], [3], [4], [5], [6], [9], [10], [14], [17], [21], [28]. Most of the work to date studies the planar case, where Γ is a collection of line segments in $\mathbb{R}^2$. Chazelle and Guibas proposed an optimal algorithm for the special case where Γ is the boundary of a simple polygon [17]. Their algorithm answers a ray-shooting query in $O(\log n)$ time using $O(n)$ space; simpler algorithms, with the same asymptotic performance bounds, were recently developed in [14] and [22]. If Γ is a collection of arbitrary segments in the plane, the best-known algorithm answers a ray-shooting query in time $O(\frac{n}{\sqrt{s}} \log^{O(1)} n)$ using $O(s^{1+\varepsilon})$ space and preprocessing[1] [1], [6], [9], where s is a parameter that varies between n and n^2. Although no lower bound is known for this case, it is conjectured that this bound is close to optimal. In spite of some recent developments, the three-dimensional ray-shooting problem seems much harder and it is still far from being fully solved. The general three-dimensional ray-shooting problem is to preprocess a collection of n triangles, so that the first triangle hit by a query ray can be computed efficiently. If the triangles are the faces of a convex polyhedron, then an optimal algorithm, with $O(\log n)$ query time and linear space, can be obtained using the hierarchical decomposition scheme of Dobkin and Kirkpatrick [20]. If the triangles form a *polyhedral terrain* (a piecewise-linear surface intersecting every vertical line in ex-

*Received by the editors February 16, 1993; accepted for publication (in revised form) July 22, 1994. A preliminary version of this paper appeared in *Proc. 4th ACM–SIAM Symp. on Discrete Algorithms*, 1993, pp. 260–270.

†Department of Computer Science, P.O. Box 90129, Duke University, Durham, NC 27708-0129. This work was supported by National Science Foundation grant CCR-91-06514.

‡School of Mathematics, Tel Aviv University, Tel Aviv 69978, Israel, and Courant Institute of Mathematical Sciences, New York University, New York, NY 10012. This work was supported by National Science Foundation grant CCR-91-22103 and by grants from the U.S.–Israeli Binational Science Foundation, the G.I.F. (the German–Israeli Foundation for Scientific Research and Development), and the Fund for Basic Research administered by the Israeli Academy of Sciences.

[1] Throughout this paper, bounds of this kind mean that, given any arbitrarily small positive constant ε, the algorithm can be fine-tuned so that its performance satisfies the bound; the multiplicative constants in such bounds usually depend on ε and tend to ∞ as $\varepsilon \downarrow 0$.

actly one point), then the technique of Chazelle et al. [15] yields an algorithm that requires $O(n^{2+\varepsilon})$ space and answers ray-shooting queries in $O(\log n)$ time. Nontrivial solutions to the general problem were obtained only recently; see [4], [6], and [10] for some of these results. The best-known algorithm for ray shooting among triangles in three dimensions is due to Agarwal and Matoušek [5]; it answers a ray-shooting query in time $O(\frac{n^{1+\varepsilon}}{s^{1/4}})$ after $O(s^{1+\varepsilon})$ space and preprocessing. The parameter s can range between n and n^4. If s assumes its maximum value, queries can be answered in $O(\log n)$ time; see [5], [6], and [28] for more details. We remark that no nontrivial lower bounds are known for the three-dimensional problem as well, although such bounds are known for the related *simplex range-searching* problem [12], which is used as a subprocedure in the solutions just mentioned.

The performance of these algorithms is rather inefficient when n is large, so a natural objective is to find special cases where this performance can be improved. The case that we consider here is where the three-dimensional scene is formed by m convex polyhedra or polyhedral terrains with a total of n faces (general nonconvex polyhedra can be decomposed into convex pieces and be replaced by these pieces). In many typical instances of the problem m is much smaller than n; for example, curved objects, like balls, cylinders, cones, etc., are usually approximated by a polyhedron with a large number of faces. Our goal is to develop an algorithm whose performance depends on both m and n, and is much better than that of the general technique when $m \ll n$.

In this paper we achieve this goal, presenting a technique that uses $O((mn)^{2+\varepsilon})$ storage and answers ray-shooting queries in $O(\log^2 n)$ time. Our algorithm is the first algorithm for ray shooting among convex polyhedra (or polyhedral terrains) whose performance depends on both m and n and matches the performance of [5] when $m \approx n$. We also present another algorithm that answers a query in time $O(m^{1/4}n^{1/2+\varepsilon})$ using $O(n^{1+\varepsilon})$ space and preprocessing, so it matches the bound of [5] when $m \approx n$, but is considerably faster when $m \ll n$.

In [7] we have presented an algorithm to preprocess a collection of m convex polygons in the plane, with a total of n vertices, into a data structure of size $O(mn \log m)$, so that a ray-shooting query can be answered in $O(\log^2 n \log^2 m)$ time. If the polygons are disjoint, or the starting point of the ray always lies in the common exterior of the polygons, then the space and preprocessing can be improved to $O((m^2 + n) \log m)$. The algorithm works even for a collection of disjoint simple polygons.

A problem related to ray shooting among a collection of convex polyhedra in three dimensions is the so-called *stabbing problem*, where one wants to determine whether a query line intersects all polyhedra. This problem seems to be easier than the ray-shooting problem: Pellegrini and Shor [29] have described a data structure of size $O(n^{2+\varepsilon})$ that can answer a stabbing query in $O(\log n)$ time.

We will first describe, in §2, the overall structure of the algorithm. We next present, in §3, an algorithm for detecting an intersection between a query segment and a collection of convex polyhedra or polyhedral terrains, which is the main subroutine used in our algorithm. For the sake of convenience, we describe the algorithm only for a collection of convex polyhedra, but the same technique works for polyhedral terrains as well. Next, in §4, we develop a variant of the technique for answering a query efficiently if only close-to-linear space is allowed. In §5, we give an application of our results to translational motion planning in $\mathbb{R}^3$: given m convex polyhedral obstacles, with a total of n faces, and a polyhedral object B, with k vertices, free to translate amidst them, we show how to preprocess them in time and space $O((kmn)^{2+\varepsilon})$, so that, given any free placement z of B and direction $\mathbf{u}$, we can compute in time $O(\log^2 kn)$ the first obstacle to be hit as B is translated from z in direction $\mathbf{u}$. Again, this is a substantial improvement over previous results when $m \ll kn$. We conclude in §6 with a discussion of our results and a few open problems.

2. The overall algorithm. Let $\mathcal{P} = \{P_1, \ldots, P_m\}$ be a set of m convex polyhedra, let n_i be the number of edges in P_i, and put $n = \sum_{i=1}^{m} n_i$ (we prefer to have n denote the total number of edges, rather than the number of faces, of the P_i's; by Euler's relation, these two quantities differ only by a small multiplicative factor). Without loss of generality assume that each face of P_i is triangulated; otherwise we can triangulate all faces of P_i by adding $O(n_i)$ additional edges. For the sake of convenience, we split the boundary of each P_i into its top portion (visible from $z = +\infty$) and its bottom portion (visible from $z = -\infty$). We construct separate data structures for the top portions and for the bottom portions, and answer a ray-shooting query by searching in both structures and by selecting the output point nearest to the ray origin. In what follows we describe only the data structure for the top portions of the given polyhedra; with a slight abuse of notation, we will refer to these top portions also as "polyhedra." We note that this step is not required if the P_i's are polyhedral terrains.

Our general ray-shooting scheme is based on the parametric searching technique of Agarwal and Matoušek [4]. In this technique we build a data structure for solving *segment-intersection detection* queries, each asking whether a query segment e intersects any of the (top portions of the) given polyhedra. Given a query ray ρ, we replace it by the segment aw, where a is the origin of ρ and w is the first point of intersection between ρ and the given polyhedra. We query the data structure with the segment aw. Of course, we do not know w (our goal is to find it!), so we feed our data structure with a generic, unspecified input w. As we will see below, each step of the algorithm asks a question of the following form: given a query point p and a hyperplane h, determine whether p lies above, below, or on h; here p is either the origin a of ρ, the Plücker point of the line containing ρ, or the generic point w. In the first two cases, we can answer the question in $O(1)$ time. To determine the position of w with respect to a plane h, it is sufficient to determine whether a and w lie on the same side of h. We compute the intersection point σ of ρ and h. If ρ does not intersect h, we can immediately conclude that a and w lie on the same side of h. Otherwise, we invoke the segment-intersection detection procedure with the segment $a\sigma$. If $a\sigma$ intersects any of the polyhedra in $\mathcal{P}$, then a and w lie on the same side of h; otherwise they lie on the opposite sides of h. This also restricts the allowed range of w. When the algorithm terminates, we obtain the exact location of w, thereby answering the original ray-shooting query. It is shown in [4] that the performance of this parametric searching technique is only slightly worse (by a logarithmic factor) than the cost of a single (explicit) segment-intersection detection query; see [4] for more details.

3. Segment-intersection detection. We now present an algorithm for the segment-intersection detection problem, i.e., preprocess $\mathcal{P}$ into a data structure, so that one can quickly determine whether a query segment intersects any of the polyhedra in $\mathcal{P}$. In this section we aim to achieve fast (polylogarithmic) query time, at the expense of storage and preprocessing. The opposite case, that of using only close-to-linear storage at the expense of query time, will be studied in §4. We will construct two data structures. The first one, denoted as $\Psi_1(\mathcal{P})$, determines whether e intersects a face of some P_i whose xy-projection does not contain any of the endpoints of the xy-projection of e. The second data structure, denoted as $\Psi_2(\mathcal{P})$, determines whether e intersects a face of some P_i whose xy-projection contains one of the endpoints of the xy-projection of e. Throughout this section we will use γ^* to denote the xy-projection of an object γ in $\mathbb{R}^3$, and A^* to denote $\{\gamma^* \mid \gamma \in A\}$ for a set A of such objects.

3.1. First data structure. In this subsection we describe a data structure $\Psi_1(\mathcal{P})$ that determines whether the query segment intersects a face of some P_i whose projection does not contain any endpoint of e^*. Let E_i denote the set of edges of P_i. We project them onto the xy-plane, and let E_i^* denote the set of resulting projected segments. Let $E^* = \bigcup_{i=1}^{m} E_i^*$. We

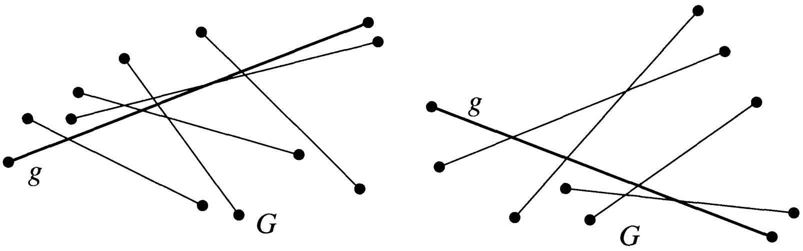


FIG. 1. *A canonical subset G of the output for a query segment g*; (i) *g intersects all segments of G from below*; (ii) *g intersects all segments of G from above.*

preprocess E^* into a data structure of size $O(n^{2+\varepsilon})$, using a variant of the technique described in Agarwal and Sharir [6], so that the set of all segments of E^* intersected by a query segment in the xy-plane can be represented as $O(\log n)$ pairwise disjoint precomputed subsets. The algorithm of [6] constructs a multilevel partition tree on E^*. Roughly speaking, it stores a family of subsets of E^*, called *canonical subsets*, into a tree-like data structure. There are at most $O((n/2^j)^{2+\varepsilon})$ canonical subsets of size between 2^{j-1} and 2^j. For a given query segment g, the canonical subsets that form the query output can be computed in $O(\log n)$ time, and there is a constant number of output subsets of size between 2^{j-1} and 2^j for each $j = 0, 1, \ldots$. Furthermore, for each canonical subset G of the query output, either the left endpoints of all segments in G lie above the line containing the query segment g, or all of them lie below that line. In the first case g, considered as a rightward-directed segment, intersects all of these segments from below, and in the second case it intersects all of them from above. (In the xy-plane, a rightward-directed segment g is said to intersect another segment e from "below" if they intersect and the left endpoint of e lies above the line containing g; intersection from "above" is defined symmetrically; see Fig. 1.) See [4] and [6] for details. In what follows we only consider the case where g intersects all segments of G from below.

Let Γ^* be a canonical subset of E^*, and let Γ be the set of corresponding pohyhedra edges, and put $\nu = |\Gamma^*|$. Let $\Gamma_i^* = \Gamma^* \cap E_i^*$, and let $\mu \leq \nu$ denote the number of nonempty Γ_i^*'s. Set $s = \lceil \nu/\mu \rceil$. We orient the edges of Γ^* from left to right. We preprocess Γ so that, for a (directed) query line ℓ in $\mathbb{R}^3$ whose xy-projection intersects all segments of Γ^*, one can quickly determine whether ℓ passes above or below the edges in Γ. The way in which we have oriented the xy-projections of the edges in Γ and of ℓ ensures that the above/below relationships between ℓ and these edges are determined solely by the sign of the *relative orientations* between ℓ and the lines containing these edges. This will enable us to determine whether a query segment intersects any of the given polyhedra, as will be described in more detail below. The relative orientation of two oriented lines ℓ, λ in $\mathbb{R}^3$ is defined to be the orientation of any simplex $abcd$, where $a, b \in \ell$, $c, d \in \lambda$, so that ℓ is oriented from a to b and λ is oriented from c to d. Equivalently, it is also the sign of the inner product between the two vectors in projective 5-space representing the *Plücker coordinates* of the two lines. (For the sake of convenience, we will not distinguish between the projective 5-space and the affine 5-space $\mathbb{R}^5$.) To be more precise, ℓ can be mapped to a point $\pi(\ell)$, called a *Plücker point*, and λ can be mapped to a hyperplane $\varpi(\lambda)$, called a *Plücker hyperplane*, in $\mathbb{R}^5$, so that ℓ has positive orientation with respect to λ if and only if $\pi(\ell)$ lies above the hyperplane $\varpi(\lambda)$. The Plücker points of all lines in $\mathbb{R}^3$ lie on a quadric surface, known as the *Plücker surface*, in $\mathbb{R}^5$. More details concerning Plücker's coordinates and relative orientations can be found in [15] and [31].

Recall that we are only preprocessing the upper portions of the polyhedra, which implies that the (relative) interiors of edges of Γ_i^* are pairwise disjoint. We define a linear ordering for a set G of t nonintersecting segments in the xy-plane: let e, e' be two segments in G; $e \prec e'$ if the x-projections of e and e' overlap and e' lies above e along a vertical line (parallel to the y-axis), if the x-projections of e and e' are disjoint and e lies to the left of e'. This is indeed a linear ordering, and it can be computed in $O(t \log t)$ time, as shown by Guibas, Overmars, and Sharir [21]. They have also shown that, for any $e, e' \in G$, if $e \prec e'$ and a rightward-directed line intersects both of them from below, then it intersects e before e'.

We sort each Γ_i^* according to this ordering and, abusing the notation slightly, we denote the resulting sequence also by Γ_i^*. Suppose $\Gamma_i^* = (e_0^*, e_1^*, \ldots, e_{t_i}^*)$. We mark the first, the last, and every sth edge of Γ_i^*, i.e., we mark e_0^*, e_s^*, $e_{2s}^*, \ldots, e_{t_i}^*$. For each marked edge e_{sj}^*, let $\Gamma(e_{sj}) = \{e_{sj+1}, e_{sj+2}, \ldots, e_u\}$, where $u = \min\{t_i, (s+1)j\}$, be the set of edges of P_i whose xy-projections form the block of edges of Γ_i^* following e_{sj}^* and ending at the next marked edge. Let G be the set of edges of polyhedra in $\mathcal{P}$ corresponding to the marked segments of Γ^*, each oriented so that its xy-projection is rightward directed; note that

$$|G| \le \sum_i \left(\left\lceil \frac{t_i}{s} \right\rceil + 1 \right) \le \sum_i \left(\frac{t_i}{s} + 2 \right) \le \frac{\mu}{\nu} (\sum_i t_i) + 2\mu \le 3\mu .$$

We will construct on G a data structure based on a partitioning scheme due to Chazelle, Sharir, and Welzl [18]. For a segment $\gamma \in \mathbb{R}^3$, this structure decomposes G further into canonical subsets, so that, for each canonical subset of G, either all the corresponding original polyhedra edges lie above γ or all of them lie below γ. Next, for each canonical subset Q in the output, we determine whether there is an edge in some $\Gamma(e)$, for $e \in Q$, that passes on the other side of γ, thereby implying that γ intersects the polyhedron P_i containing e. In more detail, this is done as follows.

Map the (directed) line containing each segment of G to its Plücker hyperplane in $\mathbb{R}^5$; let H denote the set of resulting hyperplanes, and put $t = |H| \le 3\mu$. Set r to be some sufficiently large constant. We compute a $(1/r)$-*net* R of H of size $O(r \log r)$. (We call a subset $R \subseteq H$ a $(1/r)$-net if every (relatively open) simplex intersecting more than $|H|/r$ hyperplanes of H intersects a hyperplane of R; it is well known that there exists such an R with the prescribed size.) R can be computed in $O(t)$ time if r is constant [11], [23]. We triangulate the arrangement $\mathcal{A}(R)$. Let Ξ denote the simplices of the triangulation that intersect the Plücker surface. By a result of Aronov, Pellegrini, and Sharir [8], the number of simplices in Ξ is $O(r^4 \log^5 r)$. By construction, each simplex in Ξ intersects at most t/r hyperplanes of H.

For each $\Delta \in \Xi$, let $H_\Delta \subseteq H$ denote the set of hyperplanes that intersect the interior of Δ. We also associate with Δ two other subsets U_Δ and L_Δ of H; U_Δ is the set of hyperplanes that lie fully above Δ, and L_Δ is the set of hyperplanes that lie fully below Δ.

We construct two auxiliary data structures on U_Δ and L_Δ. Let

$$\overline{U}_\Delta = \bigcup \{\Gamma(e) \mid e \text{ is an edge corresponding to a hyperplane in } U_\Delta\},$$
$$\overline{L}_\Delta = \bigcup \{\Gamma(e) \mid e \text{ is an edge corresponding to a hyperplane in } L_\Delta\}.$$

Note that $|\overline{U}_\Delta|, |\overline{L}_\Delta| \le st$. We map the lines containing the edges of $\overline{U}_\Delta$ to their Plücker hyperplanes in $\mathbb{R}^5$ and preprocess their lower envelope for point-location queries, using an algorithm of Clarkson [19]. That is, we preprocess the hyperplanes into a data structure, so that we can quickly determine whether a query point lies below all hyperplanes. Similarly, we map the edges of $\overline{L}_\Delta$ to their Plücker hyperplanes and preprocess their upper envelope for similar point location queries. Each point location structure requires $O((st)^{2+\delta})$ space and preprocessing time, for any $\delta > 0$, and answers a query in $O(\log st)$ time; see [19] for details.

Our data structure needs one more ingredient: for each simplex $\Delta \in \Xi$ and for each polyhedron P_i we consider the first (marked) edge, if any, of Γ_i that contributes a hyperplane to $U_\Delta \cup L_\Delta$. For each such edge e, let f_1, f_2 be the two faces of P_i incident to e (if e is the first or the last edge of Γ_i, it is possible that only one of these faces is defined). Let F_Δ be the collection of at most $2t$ resulting faces. We preprocess F_Δ for segment-intersection detection queries using the techniques described in [5] and [6]. It requires $O(t^{4+\delta})$ space and preprocessing time, for any $\delta > 0$, and can determine in $O(\log t)$ time whether a query segment intersects any triangle in F_Δ.

Finally, we recursively preprocess H_Δ, for each simplex $\Delta \in \Xi$. The recursion stops when $|H_\Delta|$ falls below some specified constant n_0. The resulting structure is a tree $T = T(\Gamma)$ of depth $O(\log n)$, each of whose nodes has degree at most $O(r^4 \log^5 r)$. We repeat the same procedure for all canonical subsets Γ of E. This completes the description of the data structure.

Let us analyze the space and preprocessing time of the above structure. First let us analyze the space required by the tree structure $T(\Gamma)$ constructed on a canonical subset Γ. Let $S(u)$ denote the space required by the subtree of T constructed on a set H_Δ consisting of u hyperplanes. Since the degree of each node in T is $O(r^4 \log^5 r)$, and the auxiliary structure stored at each child requires $O(s^{2+\delta}u^{2+\delta} + u^{4+\delta})$ space, we get the following recurrence:

$$S(u) \le \begin{cases} O(1) & \text{if } u \le n_0, \\ c_1 r^4 \log^5 r \cdot S\left(\frac{u}{r}\right) + c_2(s^{2+\delta}u^{2+\delta} + u^{4+\delta}) & \text{if } u > n_0, \end{cases}$$

where c_1, c_2, and n_0 are some appropriate absolute constants. The solution of this recurrence is

$$S(u) \le A s^{2+\delta'} u^{4+\delta'}, \tag{1}$$

for another $\delta' > 0$ that tends to 0 with δ and for some sufficiently large constant $A = A(\delta')$. Indeed, arguing inductively, (1) obviously holds for $u \le n_0$, and for $u > n_0$ we have

$$\begin{aligned} S(u) &\le c_1 r^4 \log^5 r \cdot S\left(\frac{u}{r}\right) + c_2(s^{2+\delta}u^{2+\delta} + u^{4+\delta}) \\ &\le c_1 r^4 \log^5 r A s^{2+\delta'} \left(\frac{u}{r}\right)^{4+\delta'} + c_2(s^{2+\delta}u^{2+\delta} + u^{4+\delta}) \\ &= A s^{2+\delta'} u^{4+\delta'} \left(\frac{c_1 \log^5 r}{r^{\delta'}} + \frac{c_2}{A}\left(\frac{(su)^{\delta-\delta'}}{u^2} + \frac{u^{\delta-\delta'}}{s^{2+\delta'}}\right)\right) \\ &\le A s^{2+\delta'} u^{4+\delta'}, \end{aligned}$$

provided that $\delta' > \delta$ and r, A are sufficiently large. Hence, the total space required by T is

$$S(t) = O(s^{2+\delta'} t^{4+\delta'}) = O(\lceil \nu/\mu \rceil^{2+\delta'} \mu^{4+\delta'}) = O(\nu^{2+\delta'} \mu^{2+\delta'}),$$

because $\mu \le \nu$. We calibrate δ so that δ' is equal to the original ε. Since there are $O\left((n/2^j)^{2+\varepsilon}\right)$ canonical subsets of E^* of size between 2^{j-1} and 2^j, the overall space required by the data structure is

$$\sum_{j=1}^{\lceil \log n \rceil} O\left(\left(\frac{n}{2^j}\right)^{2+\varepsilon}\right) \cdot O(2^{j(2+\varepsilon)} m^{2+\varepsilon}) = O((mn)^{2+\varepsilon'}),$$

for another $\varepsilon' > 0$ that tends to 0 with ε. Hence the total storage required by the data structure is $O((mn)^{2+\varepsilon})$ for any $\varepsilon > 0$. Following a similar analysis, one can show that the preprocessing time is also $O((mn)^{2+\varepsilon})$.

3.2. Second data structure. Next, we describe the second data structure $\Psi_2(\mathcal{P})$, which, given a query segment γ, determines whether there is a face f of some polyhedron P_i such that γ intersects f and that f^* contains one of the endpoints of γ^*. The data structure is again based on the partitioning scheme due to Chazelle et al. [18]. As in the preceding subsection, we consider here only the top portions of the given polyhedra. Choosing a sufficiently large constant parameter r, we partition the plane, in $O(n)$ time, into a collection Δ of $O(r^2)$ triangles, so that each triangle intersects at most n/r edges of E^* [23]. With each triangle $\Delta \in \Delta$, we associate a subset $E^*_\Delta \subseteq E^*$, and a subset F_Δ of faces of polyhedra in $\mathcal{P}$. An edge e^* belongs to E^*_Δ if the boundary of one of the triangles incident to e^* (i.e., the projected polyhedra faces) intersects Δ. A face f belongs to F_Δ if $\Delta \subset f^*$. It is easily seen that $|E^*_\Delta| \le 5n/r$ and $|F_\Delta| \le m$. We preprocess the triangles of F_Δ for segment-intersection detection queries, as in the first data structure [5], [6]. This structure requires $O(m^{4+\delta})$ space and preprocessing time, for any $\delta > 0$, and answers a query in $O(\log m) = O(\log n)$ time. We recursively preprocess each E^*_Δ and its associated collection of incident faces, and thereby obtain the entire structure. Note that each recursive processing of a set E^*_Δ involves $n_\Delta = O(n/r)$ edges, at most $2n_\Delta$ incident faces, and $m_\Delta \le n_\Delta$ polyhedra to which these edges and faces belong. $\Psi_2(\mathcal{P})$ is thus a tree of height $O(\log n)$.

If we denote by $S'(m,n)$ the maximum space required by $\Psi_2(\mathcal{P})$ for a collection of m polyhedra with n edges, then $S'(m,n)$ satisfies the following recurrence:

$$S'(m,n) \le \begin{cases} O(1) & \text{if } n \le n_0, \\ \sum_{j=1}^{c_1 r^2} S'(m_{\Delta_j}, n_{\Delta_j}) + c_2 m^{4+\delta} & \text{if } n > n_0, \end{cases}$$

where $m_{\Delta_j} \le n_{\Delta_j} \le \frac{5n}{r}$, for each j, and n_0, c_1, c_2 are appropriate constants. The solution of this recurrence is

$$S'(m,n) \le B(mn)^{2+\varepsilon}, \tag{2}$$

for some constant B and another $\varepsilon > 0$ that tends to 0 with δ. Indeed,

$$\begin{aligned} S'(m,n) &\le \sum_{j=1}^{c_1 r^2} S'(m_{\Delta_j}, n_{\Delta_j}) + c_2 m^{4+\delta} \\ &\le c_1 r^2 B \left(\frac{5mn}{r}\right)^{2+\varepsilon} + c_2 m^{4+\delta} \\ &\le B(mn)^{2+\varepsilon} \left(\frac{5^{2+\varepsilon} c_1}{r^\varepsilon} + \frac{c_2}{B} \cdot \frac{m^{\delta-\varepsilon}}{n^\varepsilon} \cdot \left(\frac{m}{n}\right)^2\right) \\ &\le B(mn)^{2+\varepsilon}, \end{aligned}$$

provided that $\varepsilon > \delta$ and r, B are chosen sufficiently large. The last inequality follows from the fact that $m \le n$. A similar analysis yields the same bound on the preprocessing time needed to construct the structure.

3.3. Answering a query. We now describe how to answer a query. Let γ be a query segment in $\mathbb{R}^3$, oriented so that its xy-projection is rightward directed, and let ℓ be the directed line containing γ. We want to determine whether γ intersects any P_i.

DEFINITION 3.1. *A segment γ lies* above *another segment e in $\mathbb{R}^3$ if their xy-projections intersect and the vertical line through this intersection meets γ at a point higher than the point at which it meets e.*

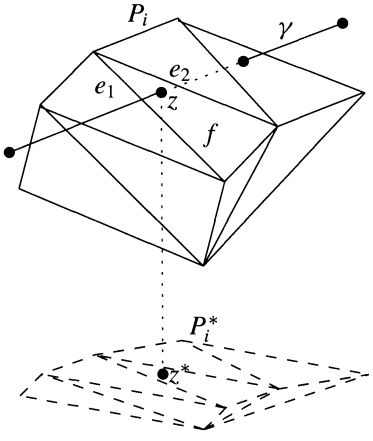


FIG. 2. *Illustration of Lemma* 3.2.

The query answering procedure is based on the following simple lemma.

LEMMA 3.2. *A segment $\gamma = pq$ in $\mathbb{R}^3$ intersects a polyhedron $P_i \in \mathcal{P}$ if and only if at least one of the following two conditions holds:*

(i) *A face of P_i, whose xy-projection contains an endpoint of γ^*, intersects γ.*

(ii) *There is a face f of P_i such that γ lies below (resp., above) one of the edges e_1 of f and lies above (resp., below) another edge e_2 of f. Moreover, assume that γ^* intersects e_2^* after e_1^*; then, for any canonical subset Γ^* of E^* that contains e_2, either e_2^* is the first edge in Γ_i^*, or there is a marked edge e^* in Γ_i^* such that $e_2 \in \Gamma(e)$, and γ lies below (resp., above) e and above (resp., below) e_2.*

Proof. The "if" part is obvious. For the "only if" part, assume that γ intersects P_i but does not intersect the faces of P_i whose projections contain the endpoints of γ^*. Let z be the leftmost intersection point of P_i and γ, and let f be the face of P_i containing z. Since the endpoints of γ^* do not lie in f^*, γ^* completely crosses f^* (see Fig. 2). Let e_1^* and e_2^* be the edges of f^* intersected by γ^* in this order. Since z is the only intersection point of γ and f, either e_1 lies below γ and e_2 lies above γ, or vice-versa.

Assume that e_2 lies above γ. Let Γ^* be a canonical subset of E^* containing e_2^*. If e_2^* is not the first edge in Γ_i^*, then let e^* be the last marked edge in Γ_i^* immediately preceding e_2^*, i.e., $e_2 \in \Gamma(e)$. Recall that the first edge of Γ_i^* is marked, so e^* is always properly defined. Since z is the leftmost intersection point of γ and P_i, and e_2 lies above γ, it is easily seen that e lies below γ. This completes the proof of the lemma. □

In view of the above lemma we can answer a query as follows. First of all, we determine, using $\Psi_2(\mathcal{P})$, whether there is a face f of some polyhedron P_i such that γ intersects f and an endpoint of γ^* lies in f^*. If we find such a face, we stop right away. In more detail, we query the structure with the left endpoint p^* of γ^*. We follow a path in $\Psi_2(\mathcal{P})$ starting from the root. At each node we do the following. If v is a leaf, we determine explicitly whether γ intersects any of the faces associated with v. Otherwise, let Δ be the set of triangles associated with v. We determine (say, by brute force) the triangle $\Delta \in \Delta$ that contains p^*. We query the auxiliary structure to determine whether any face of F_Δ intersects γ. If the answer is yes, we stop. Otherwise we descend to the child of v corresponding to the triangle Δ and recursively search within E_Δ^*. We apply the same procedure to the right endpoint of γ. The correctness of this procedure is easy to verify.

Next, we query $\Psi_1(\mathcal{P})$ with γ. First, we find all segments of E^* intersected by γ^*. Let Γ^* be a canonical subset in the query output. Without loss of generality assume that the left endpoints of all segments in Γ^* lie above γ^*, and all their right endpoints lie below γ^*, so that γ^* intersects all segments of Γ^* from below. In this case γ lies above (resp., below) a segment e of Γ (with both γ^* and e^* being rightward directed) if and only if the relative orientation of γ with respect to e is positive (resp., negative); see [15]. By Lemma 3.2, it suffices to determine whether there is a Γ_i such that either γ intersects a face incident to the first marked edge of Γ_i, or Γ_i contains a marked edge e that lies below (resp., above) γ but an edge of $\Gamma(e)$ lies above (resp., below) γ. We map the (directed) line ℓ containing γ to its Plücker point $\pi(\ell)$ in $\mathbb{R}^5$. We determine the simplex $\Delta \in \Xi$ that contains $\pi(\ell)$ (since $\pi(\ell)$ lies on the Plücker surface, Ξ contains such a simplex). First, we determine in $O(\log n)$ time whether γ intersects any triangle of F_Δ. If the answer is "yes," we stop right away. Otherwise, we continue as follows.

By construction, $\pi(\ell)$ lies above all hyperplanes of L_Δ and below all hyperplanes of U_Δ, and therefore γ lies above (resp., below) all edges e corresponding to the hyperplanes of L_Δ (resp., U_Δ). We now have to determine whether γ lies below (resp., above) any edge in $\overline{L}_\Delta$ (resp., $\overline{U}_\Delta$). To do so, we locate $\pi(\ell)$ in the upper envelope of $\overline{L}_\Delta$ (resp., in the lower envelope of $\overline{U}_\Delta$). If $\pi(\ell)$ does not lie above the upper envelope of $\overline{L}_\Delta$, then $\pi(\ell)$ lies below some hyperplane of $\overline{L}_\Delta$, which implies that γ passes below the corresponding edge, and therefore intersects one of the polyhedra. We handle $\overline{U}_\Delta$ symmetrically. This completes the description of the query answering procedure. The correctness of the procedure is easy to verify, in view of Lemma 3.2.

Let $Q(u)$ be the maximum query time spent at a subtree of T consisting of u hyperplanes. Since we spend $O(\log n)$ time in querying the auxiliary structure stored at u, we get the following recurrence:

$$Q(u) \le Q(u/r) + O(\log u). \tag{3}$$

The solution of (3) is easily seen to be $O(\log^2 u)$ if $r > 1$ is constant, so the total time spent in querying the first data structure, for a fixed canonical set Γ^* of size ν, is $O(\log^2 \nu)$. The solution of (3) can be improved to $O(\log \nu)$ by choosing $r = n^\varepsilon$. However, if $r = \nu^\varepsilon$, we cannot use a brute-force method to find the simplex of the partitioning Ξ in Plücker 5-space, which contains the query point. Instead, we preprocess the hyperplanes containing the facets of simplices in Ξ for answering point location queries using the algorithm of Chazelle and Friedman [16]. Their algorithm preprocesses a collection of n hyperplanes in $\mathbb{R}^d$ into a data structure of size $O(n^{d+\delta})$, so that a point location query can be answered in time $O(\log n)$. Thus, the simplex containing a query point can be computed in time $O(\log \nu)$ using $O((r^4 \log^5 r)^{5+\delta}) = O(r^{10+\delta}) = O(\nu^{\varepsilon(10+\delta)})$ space, for an arbitrarily small constant $\delta > 0$. It is easily seen that this additional structure does not affect the asymptotic bound on the total storage required by T. One can similarly modify the algorithm of Agarwal and Sharir [6], so that the overall query time of the first data structure, summed over all canonical subsets of E^*, also reduces to $O(\log n)$; see [10] and [27] for details. A similar analysis shows that the time spent in querying $\Psi_2(\mathcal{P})$ is $O(\log^2 n)$ if r is chosen to be a constant, and that it can be improved to $O(\log n)$ by choosing $r = n^\varepsilon$, modifying the structure as above.

Hence, we obtain the following theorem.

THEOREM 3.3. *Let $\mathcal{P} = \{P_1, \dots, P_m\}$ be a collection of m (possibly intersecting) convex polyhedra in $\mathbb{R}^3$ with a total of n edges. Given any $\varepsilon > 0$, we can preprocess $\mathcal{P}$ in time $O((mn)^{2+\varepsilon})$ into a data structure of size $O((mn)^{2+\varepsilon})$, so that an intersection between $\mathcal{P}$ and a query segment in $\mathbb{R}^3$ can be detected in $O(\log n)$ time.*

Remark 3.4. Notice that we never used the fact that the P_i's are convex polyhedra. The only property we needed was that the xy-projections of edges in each E_i were pairwise disjoint (and that the projected faces enclosed by the edges of E_i^* formed a simply connected planar region). Hence, the above algorithm also works for polyhedral terrains. We need the nonintersecting property of E_i^* to order the segments of a canonical subset Γ using the algorithm of Guibas, Overmars, and Sharir [21]. This is the only step that does not extend to arbitrary nonconvex polyhedra. We leave it to the reader to verify that our technique does indeed carry over to the case of polyhedral terrains.

We can now plug the above procedure into the general parametric searching procedure of Agarwal and Matoušek [4]. We have explained in §2 how this is done. To complete the description, one has to check that the operations that are performed by the above segment-intersection detection procedure conform to the set-up of §2. In particular, we have to ensure that we can simulate the segment-intersection detection algorithm on the segment $o\sigma$, where o is the starting point of the ray and σ is the (unknown) first intersection point of the query ray and $\mathcal{P}$, as described in §2. First of all, observe that the coordinates of the Plücker point of the line ℓ containing a query segment depend only on ℓ and not on the endpoints of γ, so the data structures constructed on Plücker hyperplanes or Plücker points can be searched explicitly, without having to generate any generic comparison, so no oracle calls are required at all. All the other data structures, constructed in two or three dimensions, are searched either with the endpoints of the query segment, or with their xy-projections, or with the point dual to the line supporting γ^*. For example, the first level of $\Psi_2(\mathcal{P})$ is searched with an endpoint of γ^*, and the first level of the two-dimensional segment-intersection structure is searched with the line supporting γ^* (actually with the point dual to that line). We leave it to the reader to verify, based on the techniques described in [6] and [5], that each comparison generated by both data structures arises in one of the following tests.

(i) Does a tetrahedron contain an endpoint of γ?

(ii) Does a triangle in the xy-plane contain an endpoint of γ^*?

(iii) Does the point dual to the line supporting γ^* lie in a given triangle in the xy-plane?

Since all of the above questions can be reduced to determining whether a given half-space contains σ, we can indeed use the general parametric search technique, as described in §2. We thus obtain the main result of the paper.

THEOREM 3.5. *Let $\mathcal{P} = \{P_1, \dots, P_m\}$ be a collection of m (possibly intersecting) convex polyhedra or polyhedral terrains in $\mathbb{R}^3$ with a total of n edges. Given any $\varepsilon > 0$, we can preprocess $\mathcal{P}$ in time $O((mn)^{2+\varepsilon})$ into a data structure of size $O((mn)^{2+\varepsilon})$, so that the first intersection point of $\mathcal{P}$ and a query ray can be computed in $O(\log^2 n)$ time.*

4. Data structures with almost linear size. In this section we consider the problem of preprocessing $\mathcal{P}$ into a data structure of size $O(n^{1+\varepsilon})$, so that a ray-shooting query can be answered in time $O(m^{1/4}n^{1/2+\varepsilon})$. We will use a similar approach to that in the preceding sections, except that we will replace each of the data structures used above by an alternative structure that uses only close-to-linear storage. As above, the structures that we will obtain are multilevel partition trees, composed of rather standard components, but, for the sake of completeness, we will provide a brief description of them. The overall structure of the algorithm is the same as in §3, that is, the algorithm uses parametric searching to replace ray-shooting queries by segment-intersection detection queries. These queries are handled, on a conceptual level, exactly as above. For the new data structures, we need the following notation.

DEFINITION 4.1. *Let S be a set of n points in $\mathbb{R}^d$ and let $r < n$ be some parameter. A* simplicial r-partition *for S is a collection $\Pi = \{(S_1, \Delta_1), \dots, (S_t, \Delta_t)\}$, where $S_1, \dots, S_t$ form a partition of S, $t \le 2r$, $|S_i| \le \lceil n/r \rceil$, and Δ_i is a simplex containing S_i. The maximum number of simplices intersected by a hyperplane is called the* crossing number *of Π.*

Matoušek has shown that there exists a simplicial r-partition with crossing number $O(r^{1-1/d})$ [25], [26]. Agarwal and Matoušek [5] proved that if S lies on an algebraic surface of some fixed degree, then the crossing number can be improved to $O(r^{1-1/(d-1)}\log^{d/(d-1)} r)$. Moreover, if $r = O(1)$, such a simplicial partition can be computed in linear time.

4.1. First data structure. We begin by describing the modified version of the first data structure, denoted as $\overline{\Psi}_1(\mathcal{P})$. Using the technique of Agarwal and Sharir [6], one can preprocess the set E of polyhedron edges into a data structure of size $O(n^{1+\varepsilon})$, consisting of $O((n/2^j)^{1+\varepsilon})$ canonical subsets of size between 2^{j-1} and 2^j, for each $j = 0, 1, \ldots$, so that all segments of E^* intersected by a query segment in the xy-plane can be reported as a collection of $O(n^{1/2+\varepsilon})$ pairwise disjoint canonical subsets; moreover, for each j there are $O((n/2^j)^{1/2+\varepsilon})$ canonical subsets of size between 2^{j-1} and 2^j in the query output. For each canonical subset in the query output, the query segment meets all its edges from below, or meets all of them from above. Let Γ be a canonical subset of this level of the data structure. We construct a secondary data structure $T(\Gamma)$ on Γ, similar to the one in the previous section. Put $\nu = |\Gamma|$ and define μ and s as in §3.1. Let G be the set of marked edges of Γ as defined in the previous section. We map the lines containing the segments of G (oriented from left to right) to their Plücker points (rather than hyperplanes as in the previous section) in $\mathbb{R}^5$. Let S be the set of resulting points; put $t = |S|$. Let r be some appropriate constant. Since all points of S lie on the (quadratic) Plücker surface, we can construct, as remarked above, a simplicial r-partition $\Pi = \{(S_1, \Delta_1), \ldots, (S_u, \Delta_u)\}$ of S with crossing number $O(r^{3/4}\log^{5/4} r)$. Let $t_i = |S_i|$, for $i = 1, \ldots, u$. We construct a secondary structure for each S_i, of the form described below, and then preprocess each pair (S_i, Δ_i) recursively. The resulting structure is a two-level partition tree of depth $O(\log n)$. Let $E(S_i)$ denote the edges of E corresponding to the points in S_i, and let

$$\overline{U}_i = \bigcup\{\,\Gamma(e) \mid e \in E(S_i)\,\}.$$

Put $\overline{u}_i = |\overline{U}_i| \leq st_i$. We map each line containing an edge of $\overline{U}_i$ to its Plücker point in $\mathbb{R}^5$ and preprocess the set of resulting points into a linear-size data structure that answers empty half-space queries (given a query half-space g in $\mathbb{R}^5$, it determines whether g contains any point of $\overline{U}_i$) in time $O(\overline{u}_i^{1/2+\delta})$, for some $0 < \delta < \varepsilon$; see [24]. We also preprocess the triangles incident to the edges in $E(S_i)$ for segment-intersection detection queries, using the algorithm of Agarwal and Matoušek [5]; it requires $O(t_i^{1+\delta})$ space and answers a query in time $O(t_i^{3/4+\delta})$. This completes the description of the first data structure. Since a secondary structure constructed on u points requires $O(u^{1+\delta})$ space, the total space required by the first data structure for a fixed canonical subset Γ with ν edges is $O(\nu^{1+\delta})$. Summing it over all canonical subsets of Γ^*, we obtain

$$\sum_{j=1}^{\lceil\log n\rceil}\left(\frac{n}{2^j}\right)^{1+\varepsilon}\cdot O(2^{j(1+\delta)}) = O(n^{1+\varepsilon})\sum_{j=1}^{\lceil\log n\rceil}2^{j(\delta-\varepsilon)} = O(n^{1+\varepsilon}),$$

since $\delta < \varepsilon$. The preprocessing time is also $O(n^{1+\varepsilon})$.

4.2. Second data structure. Next, we describe the modified version of the second data structure, denoted as $\overline{\Psi}_2(\mathcal{P})$. We will construct a four-level partition tree. The first three levels of $\overline{\Psi}_2(\mathcal{P})$ will filter out the faces of polyhedra in $\mathcal{P}$ whose xy-projections contain an endpoint of γ, and the fourth level will determine whether any of these faces intersect the query segment.

In more detail, let F be the set of faces of polyhedra in $\mathcal{P}$, and let F^* denote the set of the xy-projections of these faces. By our assumption, each face in F^* is a triangle. We split each face $f \in F$ into two subtriangles by drawing a plane parallel to the yz-plane through the "middle" vertex of f, as shown in Fig. 3. We will continue to denote the new set of faces by

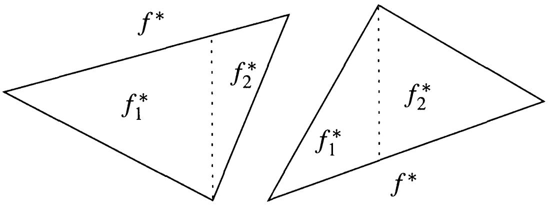


FIG. 3. *Splitting each triangle into two triangles.*

F and the set of their xy-projections by F^*. Each triangle in F^* has one vertical edge (i.e., parallel to the y-axis) and two nonvertical edges. Let I denote the set of the x-projections of triangles in F^*. We construct a segment tree B on I; see [30] for details on segment trees. Every node v of B is associated with an interval δ_v and stores a "canonical" subset $I(v)$ of I, where each interval in $I(v)$ contains δ_v (but does not contain $\delta_{p(v)}$, where $p(v)$ is the parent of v). Moreover, $\sum_{v\in B}|I(v)| = O(n\log n)$. We preprocess each canonical subset separately.

Let $F^*(v)$ denote the set of triangles corresponding to the intervals in $I(v)$; put $t = |F^*(v)|$. We construct a partition tree $\mathcal{T} = \mathcal{T}(v)$ on $F^*(v)$. For each triangle $\triangle \in F^*(v)$, pick one of its nonvertical edges. Let V denote the set of points in the xy-plane, dual to the lines supporting these edges. Each node w of $\mathcal{T}$ will be associated with a subset of V and a triangle. The root u is associated with V and the entire xy-plane. Let r be some appropriate constant. We construct, in linear time, a simplicial r-partition $\Pi = \{(V_1, \tau_1), \ldots, (V_u, \tau_u)\}$ for V with crossing number $O(\sqrt{r})$ [25], [26]. We create a child w_i of u corresponding to each pair (V_i, τ_i) and store a two-level auxiliary structure at w_i, as detailed below. We recursively preprocess V_i and attach the resulting tree of auxiliary substructures to w_i. The recursion stops when the number of points in V_i falls below some prespecified constant.

The auxiliary structure at w_i is constructed as follows. For each point $p \in V_i$, we pick the other nonvertical edge of the triangle corresponding to p. Let W_i denote the set of points dual to lines supporting these edges; $|W_i| \le \lceil t/r \rceil$. We construct a partition tree on W_i as above. The root of the partition tree is associated with W_i and the entire xy-plane. We compute a simplicial r-partition $\Pi_i = \{(W_{i1}, \tau_{i1}), \ldots, (W_{iu_i}, \tau_{iu_i})\}$ for W_i, create a child w_i of the root for each pair (W_{ij}, τ_{ij}), and recursively preprocess W_{ij}. Let F_{ij} denote the set of faces in $F(v)$ corresponding to points in W_{ij}. If F_{ij} contains two faces of the same polyhedron, we do not store any structure at w_i (because, for any projected endpoint p of a query segment, any set F_{ij} that will be picked up by the query will have the property that all its projected triangles contain p, so an F_{ij} of the above kind will never have to be processed by any query). Otherwise we preprocess F_{ij} for segment-intersection detection queries in 3-space, using the algorithm of Agarwal and Matoušek [5], and store it at the node corresponding to (W_{ij}, τ_{ij}) as its auxiliary structure. This completes the description of the second data structure. Following the same analysis as for $\overline{\Psi}_1(\mathcal{P})$, one can show that the total space and preprocessing time required are $O(n^{1+\varepsilon})$.

4.3. Answering a query. A segment-intersection detection query is answered exactly the same way as in §3.3. That is, we first query $\overline{\Psi}_2(\mathcal{P})$ and determine whether any of the faces, whose xy-projections contain an endpoint of γ, intersects γ. If γ does not intersect any such face, then we determine, using $\overline{\Psi}_1(\mathcal{P})$, whether e intersects any other face of the polyhedra in $\mathcal{P}$.

Let p be the left endpoint of γ^*. We want to determine whether any face whose xy-projection contains p intersects γ. A triangle $f^* \in F^*$ contains p if and only if the following three conditions are satisfied:

(i) the x-projection of f^* contains the x-coordinate of p,
(ii) one of the nonvertical edges of f^* lies above p, and
(iii) the other nonvertical edge of f^* lies below p (in the xy-plane).

To filter out the faces whose xy-projections satisfy these three conditions, we query the segment tree B (the first-level structure of $\overline{\Psi}_2(\mathcal{P})$) with p and compute the $O(\log n)$ nodes of B whose associated intervals contain the x-coordinate of p. Let v be such a node of B, and let ℓ denote the line dual to p. We query the (second-level) partition tree $\mathcal{T} = \mathcal{T}(v)$ with ℓ. We start at the root and at each node w, visited by the algorithm, we do the following. Let (V, τ) be the pair associated with w. If w is a leaf, we directly determine whether γ intersects any of the triangles of F corresponding to points in V. So assume that w is an internal node. If τ intersects ℓ, we visit all the children of w. Otherwise, we visit the auxiliary structure stored at w. Without loss of generality assume that τ lies fully above ℓ; the other case can be handled symmetrically. We search the (third-level) auxiliary partition tree stored at w with ℓ in the same way as we searched $\mathcal{T}(v)$. That is, at each node ξ of this third-level structure, we do the following. Let (W', τ') be the pair associated with ξ. If ℓ intersects the triangle τ', we recursively search at each child of ξ. If τ' lies fully above ℓ, then both nonvertical edges of triangles in $F^*_{\tau'}$, the set of triangles of $F^*(v)$ corresponding to the points in W', lie above the left endpoint of γ^*, which implies that p lies below all triangles in $F^*_{\tau'}$. Consequently, we do not search the subtree rooted at ξ any further. Finally, if τ' lies fully below ℓ, we can conclude that p lies in the triangles of $F^*_{\tau'}$. Let $F_{\tau'} \subseteq F(v)$ denote the set of faces of polyhedra corresponding to triangles in $F^*_{\tau'}$. Using the fourth-level auxiliary structure stored at ξ, we test whether γ intersects any of the triangles in $F_{\tau'}$. If γ intersects a triangle of $F_{\tau'}$, we stop right away, otherwise we continue with the overall search. If no face of $F(v)$, whose xy-projection contains p, intersects γ, we repeat the above step with the right endpoint of γ.

If the above procedure does not detect an intersection between $\mathcal{P}$ and γ, we query $\overline{\Psi}_1(\mathcal{P})$ as follows. We determine the segments of E^* intersected by the xy-projection γ^* of the query segment γ. Let Γ be a canonical subset of the output to this subquery. We map the line containing γ to its Plücker hyperplane h, and query the secondary structure (partition tree) constructed on the marked edges of Γ with h. The root of the partition tree stores a simplicial r-partition $\Pi = \{(S_1, \Delta_1), \ldots, (S_\alpha, \Delta_\alpha)\}$. For each simplex Δ_i in Π, we test whether h intersects Δ_i. If h intersects Δ_i, we recursively search the substructure constructed on S_i. On the other hand, if h does not intersect Δ_i, all points in S_i lie either above h or below h, say above h. Then, as is easily checked, we know that all edges $e \in E(S_i)$ lie above γ, so it suffices to determine whether γ intersects any of the faces incident to any edge in $E(S_i)$, or whether any edge of $\overline{U}_i$ lies below γ. Both of these conditions can be tested, by querying the third-level substructures stored with $E(S_i)$, in time $O(u^{3/4+\delta} + s\overline{u}^{1/2+\delta})$ (where $u = |S_i|$ and $su \geq |\overline{U}_i|$). This completes the description of the algorithm for answering a query.

We now analyze the total time spent in answering a query. First let us consider the time spent in querying $\overline{\Psi}_2(\mathcal{P})$. For $1 \leq i \leq 4$, let $Q^{(i)}(m, n)$ denote the maximum query time at an ith level structure (including the time spent at its auxiliary structures), storing n triangles which belong to m different polyhedra. Since the above procedure visits only $O(\log n)$ nodes of the segment tree,

$$Q^{(1)}(m, n) = O(\log n) \cdot Q^{(2)}(m, n). \tag{4}$$

The fourth-level structure of $\overline{\Psi}_2(\mathcal{P})$ has at most one triangle from each polyhedron, so $m = n$ and, by [5], $Q^{(4)}(m, n) = O(m^{3/4+\delta})$, for any $\delta > 0$. Finally, for $i = 2, 3$ (i.e., for partition trees constructed on sets of nonvertical edges of triangles), a line intersects only $O(\sqrt{r})$ triangles of the r-partition constructed on the set of points associated with any tree node. Therefore the query line recursively searches only $O(\sqrt{r})$ children of any interior node, which yields the following recurrence:

$$Q^{(i)}(m,n) \le \begin{cases} \sum_{i=1}^{c_1\sqrt{r}} Q^{(i)}(m_i,n_i) + Q^{(i+1)}(m,n) & \text{if } n \ge n_0, \\ O(1) & \text{if } n < n_0, \end{cases} \tag{5}$$

where n_0, c_1 are appropriate constants, and $n_i \le n/r$, $m_i \le m$ for each i. We claim that the solution of the above recurrence is

$$Q^{(i)}(m,n) \le Am^{1/4}n^{1/2+\varepsilon}, \tag{6}$$

for any $\varepsilon > 0$. We will prove the recurrence for $i = 3$; a similar proof works for $i = 2$. Equation (6) is obviously true for $n \le n_0$ (provided A is chosen sufficiently large); and for $n > n_0$, we have, for an appropriate constant $c_2 > 0$,

$$\begin{aligned} Q^{(3)}(m,n) &\le \sum_{i=1}^{c_1\sqrt{r}} Q^{(3)}(m_i,n_i) + c_2m^{3/4+\delta} \\ &\le \sum_{i=1}^{c_1\sqrt{r}} Am_i^{1/4}\left(\frac{n}{r}\right)^{1/2+\varepsilon} + c_2m^{3/4+\delta} \\ &\le Am^{1/4}n^{1/2+\varepsilon}\frac{c_1}{r^{\varepsilon}} + c_2m^{3/4+\delta} \\ &\le Am^{1/4}n^{1/2+\varepsilon}, \end{aligned}$$

provided that $\varepsilon \ge \delta$ and that the constants r, A are chosen sufficiently large (we also use here the obvious fact that $m \le n$).

Similarly, we can show that $Q^{(2)}(m,n) = O(m^{1/4}n^{1/2+\varepsilon})$. Plugging (6) into (4), we can conclude that the maximum query of $\Psi_2(\mathcal{P})$ is $O(m^{1/4}n^{1/2+\varepsilon})$.

Next, we analyze the query time of $\Psi_1(\mathcal{P})$. Recall that we spend $O(u^{3/4+\delta} + (su)^{1/2+\delta})$ time at a third-level substructure of size u. Let $Q^{(2)}(u)$ denote the maximum query time of the second-level partition constructed on a subset of u marked edges of Γ. Since the crossing number of the simplicial r-partition stored at each node of the (second-level) partition tree is $O(r^{3/4}\log^{5/4} r)$, we obtain the following recurrence:

$$Q^{(2)}(u) \le c_1r^{3/4}\log^{5/4} r \cdot Q^{(2)}\left(\frac{u}{r}\right) + c_2(u^{3/4+\delta} + (su)^{1/2+\delta}), \tag{7}$$

where c_1, c_2 are appropriate constants. The solution of the above recurrence is easily checked to be

$$Q^{(2)}(u) = O(s^{1/2+\delta}u^{3/4+\delta}) .$$

Hence, the total time spent in querying a single first-level canonical subset Γ is

$$O(s^{1/2+\delta}\mu^{3/4+\delta}) = O\left(\left\lceil\frac{\nu}{\mu}\right\rceil^{1/2+\delta}\mu^{3/4+\delta}\right) = O(\mu^{1/4}\nu^{1/2+\delta}).$$

Recall that there are $O((n/2^j)^{1+\varepsilon})$ first-level canonical subsets of size between 2^{j-1} and 2^j, so the overall query time of the first data structure is at most

$$\sum_{j=1}^{\lceil\log n\rceil} O\left(\left(\frac{n}{2^j}\right)^{1+\varepsilon}\right) \cdot O\left(m^{1/4}2^{j(1/2+\delta)}\right) = O(m^{1/4}n^{1/2+\varepsilon'}),$$

where $\varepsilon' \geq \varepsilon + \delta$ is another arbitrarily small constant. Putting everything together, we obtain the following theorem.

THEOREM 4.2. *Let $\mathcal{P} = \{P_1, \ldots, P_m\}$ be a collection of m (possibly intersecting) convex polyhedra or polyhedral terrains in $\mathbb{R}^3$ with a total of n edges. Given any $\varepsilon > 0$, we can preprocess $\mathcal{P}$ in time $O(n^{1+\varepsilon})$ into a data structure of size $O(n^{1+\varepsilon})$, so that one can determine in time $O(m^{1/4}n^{1/2+\varepsilon})$ whether a query segment intersects any polyhedron (or polyhedral terrain) in $\mathcal{P}$.*

Again, we plug this procedure into the parametric search technique to answer a ray-shooting query. We leave it to the reader to verify that each comparison can be reduced to determining whether the first intersection of the ray and $\mathcal{P}$ lies in a query half-space. Hence, we can conclude with the following theorem.

THEOREM 4.3. *Let $\mathcal{P} = \{P_1, \ldots, P_m\}$ be a collection of m (possibly intersecting) convex polyhedra or polyhedral terrains in $\mathbb{R}^3$ with a total of n edges. Given any $\varepsilon > 0$, we can preprocess $\mathcal{P}$ in time $O(n^{1+\varepsilon})$ into a data structure of size $O(n^{1+\varepsilon})$, so that the first intersection point, if any, of a query ray with the polyhedra of $\mathcal{P}$ can be computed in $O(m^{1/4}n^{1/2+\varepsilon})$ time.*

5. Application to motion planning. An interesting application of our algorithm for ray shooting amidst convex polyhedra is the following motion-planning problem in $\mathbb{R}^3$. Suppose we have a convex polyhedral object B bounded by k faces, which is free to translate amidst a collection of m convex polyhedral obstacles, $A_1, \ldots, A_m$, with a total of n faces. Preprocess them into a data structure so that, given any free placement Z of B and a direction $\mathbf{u}$, we can efficiently find the first obstacle, if any, to be hit as we translate B from Z in direction $\mathbf{u}$.

This problem can be easily reduced to the ray-shooting problem amidst a collection of *intersecting* convex polyhedral objects. We simply compute the Minkowski differences (also known as *expanded obstacles*)

$$A_i^* = A_i - B_0 = \{x - y \mid x \in A_i,\ y \in B_0\},$$

for $i = 1, \ldots, m$; here B_0 denotes some standard placement of the object B. If A_i has n_i faces, then A_i^* is a convex polyhedron consisting of at most $O(kn_i)$ faces, so the total number of faces of the expanded obstacles is $O(kn)$.

Now, given a free placement Z of B and a direction $\mathbf{u}$, we can find the first obstacle to be hit by B when it is translated from Z in direction $\mathbf{u}$ by performing a ray-shooting query with the ray $(z, \mathbf{u})$ amidst the expanded obstacles A_i^*, where z is the displacement of B from its standard placement B_0 to the placement Z. The first expanded obstacle that the ray hits corresponds to the first obstacle that B hits. Applying the results in the previous sections, we thus obtain the following corollary.

COROLLARY 5.1. *Given a collection of m convex polyhedral obstacles with a total of n faces and a convex polyhedral object B with k faces, we can preprocess them, in time $O((kmn)^{2+\varepsilon})$, into a data structure of size $O((kmn)^{2+\varepsilon})$, so that, given any placement Z of B and direction $\mathbf{u}$, we can determine, in $O(\log^2 kn)$ time, the first obstacle, if any, that B hits when translated from Z in direction $\mathbf{u}$.*

6. Conclusion. In this paper we presented two data structures for answering ray-shooting queries among a collection of m convex polyhedra or polyhedral terrains with a total of n faces. The first method answers a query in $O(\log^2 n)$ time using $O((mn)^{2+\varepsilon})$ space and preprocessing time, while the second method achieves $O(m^{1/4}n^{1/2+\varepsilon})$ query time, using $O(n^{1+\varepsilon})$ space and preprocessing time. When $m \ll n$, both methods are significantly better than previous techniques, which either require $O(n^{4+\varepsilon})$ space and preprocessing for a polylogarithmic query time, or require $O(n^{3/4+\varepsilon})$ query time for almost-linear storage and preprocessing. Of course, when $m \approx n$, our algorithms perform as well as the previous ones. For $m = 1$, the performance of our algorithms matches that of the best known algorithm for ray shooting in a polyhedral terrain [4], [15], but is not as good as the best known technique for a single convex polyhedron.

We conclude by mentioning some open problems:

1. If $m = 1$, neither of our structures achieves close-to-optimal performance for the case of a single convex polyhedron. For example, the space and preprocessing time of our first technique are $O(n^{2+\varepsilon})$, in contrast with the technique of [20], which can answer a ray-shooting query in $O(\log n)$ time, using $O(n)$ space and $O(n \log n)$ preprocessing. It is clear from this that we are not fully exploiting the fact that the given polyhedra are convex. It would be interesting to improve our techniques further so that their performance approaches that of [20] for the case of a single convex polyhedron. (We are also not exploiting at all the pairwise disjointness of the given polyhedra.) A plausible goal to shoot for might be to improve our first technique so that it yields a data structure that requires only $O(m^{4+\varepsilon} + n)$ or $O(m^{3+\varepsilon} n)$ space and preprocessing.
2. How far can our techniques be extended? It seems unlikely that any improvement over the general previous techniques can be obtained for ray shooting among arbitrary nonconvex polyhedra, but perhaps there are useful special cases, beyond the case of terrains, for which faster techniques exist.
3. One application of our algorithms is for the case of a small number of curved surfaces, each approximated by a polyhedral surface with a large number of faces. An alternative attack on this case would be to drop the polyhedral representation altogether and to develop special techniques for ray shooting amidst curved objects. This is a more difficult problem for general surfaces, although some progress has recently been done for the case of ray shooting amidst spheres (see [2] and [5]).
4. Finally, as mentioned in the introduction, no nontrivial lower bounds are known for any of the ray-shooting problems.

Acknowledgments. The authors thank Mark de Berg and other attendants of the 2nd Utrecht Workshop on Computational Geometry and Its Application for raising the problem studied here.

REFERENCES

[1] P. K. Agarwal, *Ray shooting and other applications of spanning trees with low stabbing number*, SIAM J. Comput., 21 (1992), pp. 540–570.

[2] P. K. Agarwal, L. Guibas, M. Pellegrini, and M. Sharir, manuscript, 1993.

[3] P. K. Agarwal, M. van Kreveld, and M. Overmars, *Intersection queries for curved objects*, J. Algorithms, 15 (1993), pp. 229–266.

[4] P. K. Agarwal and J. Matoušek, *Ray shooting and parametric search*, SIAM J. Comput., 22 (1993), pp. 794–806.

[5] ———, *Range searching with semi-algebraic sets*, Discrete Comput. Geom., 11 (1994), pp. 393–418.

[6] P. K. Agarwal and M. Sharir, *Applications of a new space partitioning technique*, Discrete Comput. Geom., 9 (1993), pp. 11–38.

[7] ———, *Ray shooting amidst convex polygons in 2D*, J. Algorithms, to appear.

[8] B. Aronov, M. Pellegrini, and M. Sharir, *On the zone of a surface in a hyperplane arrangement*, Discrete Comput. Geom., 9 (1993), pp. 177–188.

[9] R. Bar Yehuda and S. Fogel, *Variations on ray shooting*, Algorithmica, 11 (1994), pp. 133–145.

[10] M. De Berg, D. Halperin, M. Overmars, J. Snoeyink, and M. Van Kreveld, *Efficient ray shooting and hidden surface removal*, Algorithmica, 12 (1994), pp. 30–53.

[11] B. Chazelle, *Cutting hyperplanes for divide-and-conquer*, Discrete Comput. Geom., 10 (1993), pp. 145–158.

[12] ———, *Lower bounds on the complexity of polytope range searching*, J. Amer. Math. Soc., 2 (1989), pp. 637–666.

[13] B. Chazelle and H. Edelsbrunner, *An optimal algorithm for intersecting line segments in the plane*, J. ACM, 39 (1992), pp. 1–54.

[14] B. Chazelle, H. Edelsbrunner, M. Grigni, L. Guibas, J. Hershberger, M. Sharir, and J. Snoeyink, *Ray shooting in polygons using geodesic triangulations*, Algorithmica, 12 (1994), pp. 54–68.

[15] B. Chazelle, H. Edelsbrunner, L. Guibas, M. Sharir, and J. Stolfi, *Lines in space: Combinatorics and algorithms*, Tech. rep. 491, Dept. of Computer Science, New York University, February 1990; Algorithmica, to appear.

[16] B. Chazelle and J. Friedman, *A deterministic view of random sampling and its use in geometry*, Combinatorica, 10 (1990), pp. 229–249.

[17] B. Chazelle and L. Guibas, *Visibility and intersection problems in plane geometry*, Discrete Comput. Geom., 4 (1989), pp. 551–589.

[18] B. Chazelle, M. Sharir, and E. Welzl, *Quasi-optimal upper bounds for simplex range searching and new zone theorems*, Algorithmica, 8 (1992), pp. 407–430.

[19] K. L. Clarkson, *A randomized algorithm for closest-point queries*, SIAM J. Comput., 17 (1988), pp. 830–847.

[20] D. Dobkin and D. Kirkpatrick, *Determining the separation of preprocessed polyhedra: A unified approach*, Proc. 17th Internat. Colloq. Automata, Languages and Programming, 1991, pp. 400–413.

[21] L. Guibas, M. Overmars, and M. Sharir, *Ray shooting, implicit point location, and related queries in arrangements of segments*, Tech. Rep. 433, Dept. of Computer Science, New York University, March 1989.

[22] J. Hershberger and S. Suri, *A pedestrian approach to ray shooting: Shoot a ray, take a walk*, J. Algorithms, 18 (1995), pp. 403–431.

[23] J. Matoušek, *Approximations and optimal geometric divide-and-conquer*, J. Comput. System Sci., 50 (1995), pp. 203–208

[24] ———, *Reporting points in halfspaces*, Computational Geometry: Theory and Applications, 2 (1992), pp. 169–186.

[25] ———, *Efficient partition trees*, Discrete Comput. Geom., 8 (1992), pp. 315–334.

[26] ———, *Range searching with efficient hierarchical cuttings*, Discrete Comput. Geom., 10 (1993), pp. 157–182.

[27] K. Mehlhorn, *Data Structures and Algorithms, III. Multi-Dimensional Searching and Computational Geometry*, Springer-Verlag, Berlin, Heidelberg, New York, 1985.

[28] M. Pellegrini, *Ray shooting in triangles in* 3*-space*, Algorithmica, 9 (1993), pp. 471–494.

[29] M. Pellegrini and P. Shor, *Finding stabbing lines in* 3*-space*, Discrete Comput. Geom., 8 (1992), pp. 191–208.

[30] F. Preparata and M. Shamos, *Computational Geometry: An Introduction*, Springer-Verlag, Heidelberg, 1985.

[31] D. M. H. Sommerville, *Analytical Geometry in Three Dimensions*, Cambridge University Press, Cambridge, UK, 1951.

SIAM J. COMPUT.
Vol. 25, No. 1, pp. 117–132, February 1996

005

A NEW CHARACTERIZATION OF TYPE-2 FEASIBILITY*

B. M. KAPRON† AND S. A. COOK‡

Abstract. K. Mehlhorn introduced a class of polynomial-time-computable operators in order to study poly-time reducibilities between functions. This class is defined using a generalization of A. Cobham's definition of feasibility for type-1 functions to type-2 functionals. Cobham's feasible functions are equivalent to the familiar poly-time functions. We generalize this equivalence to type-2 functionals. This requires a definition of the notion "poly time in the length of type-1 inputs." The proof of this equivalence is not a simple generalization of the proof for type-1 functions; it depends on the fact that Mehlhorn's class is closed under a strong form of simultaneous limited recursion on notation and requires an analysis of the structure of oracle queries in time-bounded computations.

Key words. type-2 computability, polynomial time, notational recursion, oracle Turing machine

AMS subject classifications. 68Q05, 68Q15, 03D65, 03D20

1. Introduction. A *type*-1 *function* is a mapping from $\mathbb{N}$ to $\mathbb{N}$. We will denote the set of all functions by ${}^{\mathbb{N}}\mathbb{N}$. A *type*-2 *functional* is a mapping from $({}^{\mathbb{N}}\mathbb{N})^k \times \mathbb{N}^l$ to $\mathbb{N}$, for some k and l. More specifically, we will call a mapping of this sort a *functional with rank* (k, l).

For type-1 functions, there is a well-established notion of computational feasibility. Namely, a function is feasible if it is computable in polynomial time on a Turing machine. More specifically, a function f is poly time if there is a Turing machine (TM) M and a polynomial p such that for all x, M with input x computes $f(x)$ and runs in time $p(n)$, where $n = |x|$, and for $x \in \mathbb{N}$, $|x|$ denotes the length of the binary notation of x, that is $\lceil \log(x+1) \rceil$. This notion of feasibility is robust in the sense that it is independent of the computational model used, assuming that the model is "reasonable." In [1], Cobham presented a machine-independent characterization of computational feasibility, via an inductive definition. Cobham's definition, while important, lacks the intuitive appeal of the machine-based characterization because, intuitively, feasibility depends on a notion of bounding computational resources (in this case running time) in a general computational model in some natural way.

Questions about feasibility arise when dealing with type-2 functionals as well, for example, in the study of reducibilities [9], computable analysis [5], and descriptive set theory [10]. Mehlhorn's study [9] of feasible reducibilities appears to be one of the first to consider the notion of feasibility for type-2 functionals. Here, a class of *poly-time operators* is defined, using a generalization of Cobham's definition. Subsequent studies, such as [10] and [4], take Mehlhorn's approach. The work done to date in this area does not address the question of whether there is a natural machine-based definition of Mehlhorn's class. In this paper, we provide an affirmative answer to the question.

2. A computational model for functionals. Our model for type-2 computability is a generalization of the familiar multitape oracle Turing machine (OTM). However, we allow arbitrary type-1 functions as oracles, rather than subsets of $\mathbb{N}$. Note that we also use the term oracle Turing machine to refer to this modified model. In addition to the normal work tapes, there is an *oracle-query tape* and an *oracle-answer tape* for each function input. These tapes are infinite in one direction. In order to query a function oracle at an input x, we write x (in binary) on the corresponding query tape, move the read head on the oracle tape to the

*Received by the editors February 23, 1994; accepted for publication (in revised form) July 25, 1994. A preliminary version of this paper appeared as [7].

†Department of Computer Science, University of Victoria, Victoria, BC V8W 3P6, Canada (bmkapron@csr.uvic.ca). The work of this author was completed while at the University of Toronto as part of his Ph.D. thesis [6].

‡Department of Computer Science, University of Toronto, Toronto, ON M5S 1A4, Canada (sacook@theory.toronto.edu).

beginning, and enter a query state for that oracle. In the next step, the value of the function at the specified input is written (in binary) at the beginning of the corresponding answer tape, and the head of the answer tape is returned to the leftmost position. The rest of the answer tape is overwritten with blanks. There is also a special, read-only *input tape*. One work tape is specified as the *output tape*.

An OTM M computes a functional F_M of rank (k, l) if it has k oracle-query states, and for all $f_1, \ldots, f_k$ and $x_1, \ldots, x_l$, whenever M is started with $x_1, \ldots, x_l$ written in binary (and separated by blanks) on its input tape, and when f_i is the function associated with query state i, M halts with $F_M(\vec{f}, \vec{x})$ written at the beginning of its output tape, followed by blanks and with the read head of the work tape in the leftmost position. In this case we say that M has *function inputs* $f_1, \ldots, f_k$ and *number inputs* $x_1, \ldots, x_l$ and that M is a *rank-*(k, l) *OTM*.

The running time of a Turing machine is normally just the number steps that it executes before halting. This is also the case for OTMs with set oracles. With function oracles, on the other hand, there are two possible conventions for the cost of an oracle call. The first is to charge one time step, reflecting our intuition that oracles are like subroutines with unlimited power. However, it is also reasonable to charge the length of the value returned by the oracle, reflecting the fact that the answer returned by the oracle must still be written down on the output tape. More formally, if we query oracle f at input $\vec{x}$, the associated cost is $\max\{1, |f(\vec{x})|\}$. Thus we have an I/O cost associated with an oracle call. We choose the latter convention.[1]

DEFINITION 2.1. *The* running time *of an OTM with a given input is the sum of the costs of the steps it executes. We denote by* $T_M(\vec{f}, \vec{x})$ *the running time of* M *on inputs* $\vec{f}$ *and* $\vec{x}$.

Because of the way we charge for oracle calls, the number of steps in a computation is not equal to its running time, as oracle calls are atomic steps with nonunit cost. We denote by $\text{Steps}(M, \vec{f}, \vec{x}, t)$ the least number of steps that M must execute on inputs $\vec{f}, \vec{x}$ so that the sum of the costs of those steps is at least t. In the case that $t > T_M(\vec{f}, \vec{x})$, we adopt the convention that $\text{Steps}(M, \vec{f}, \vec{x}, t)$ denotes $\text{Steps}(M, \vec{f}, \vec{x}, T_M(\vec{f}, \vec{x})) + 1$. We will write $\text{Steps}(t)$ when $M, \vec{f}$, and $\vec{x}$ are understood. We will also denote by $S_M(\vec{f}, \vec{x})$ the value $\text{Steps}(T_M(\vec{f}, \vec{x}))$, that is, the total number of steps taken by M on inputs $\vec{f}$ and $\vec{x}$ before halting. It is important to note that the computation of an OTM for a given function input depends only on the values of the function at those points which are actually queried during the computation. This is formalized as follows (we will restrict our attention to rank-$(1, 1)$ functionals for the sake of simplicity.)

DEFINITION 2.2. *For any function* f, *any rank-*$(1, 1)$ *OTM* M, *and any* $t, x \in \mathbb{N}$, *let* $Q(M, f, x, t)$ *denote the* query set *consisting of all* y *such that* M *with inputs* f *and* x *queries* f *at* y *within* $\text{Steps}(t)$ *steps of its execution. For any set* $Q \subseteq \mathbb{N}$ *and any function* f, *let* f_Q, *the* query restriction of f, *be the function such that* $f_Q(y) = f(y)$ *for all* $y \in Q$, *and* $f_Q(y) = 0$ *otherwise.*

PROPOSITION 2.3. *If* $Q = Q(M, f, x, t)$, *then the first* $\text{Steps}(t)$ *steps of the execution of* M *on inputs* f *and* x *are identical to its first* $\text{Steps}(t)$ *steps on inputs* f_Q *and* x.

3. Basic feasible functionals. Cobham [1] gave an inductive definition of type-1 feasible functions in terms of certain initial functions and closure conditions. The most important aspect of this definition is closure under *limited recursion on notation*. Cobham's feasible functions coincide exactly with the familiar poly-time functions. Mehlhorn [9] generalized Cobham's definition to type-2 functionals to define the class of *polynomial-time operators*. We will consider a functional version of this generalization based on that given by Townsend [10]. We differ somewhat from Townsend, who considers functionals over $\{0, 1\}^*$. Also, we include all

[1] Recently, A. Ignjatovič [6] showed that the main result of this paper also holds in the unit cost model. His techniques differ significantly from ours.

type-1 poly-time functions as initial functionals. This simplifies the closure schemes needed for argument manipulation. Note that we will refer to functionals in this class as *basic feasible functionals* (BFFs) rather than poly-time functionals. An explanation of this terminology is given in [4]. We first introduce some schemes for defining functionals.

DEFINITION 3.1. *F is defined from $H, G_1, \ldots, G_l$ by* functional composition *if for all $\vec{f}$ and $\vec{x}$,*

$$F(\vec{f}, \vec{x}) = H(\vec{f}, G_1(\vec{f}, \vec{x}), \ldots, G_l(\vec{f}, \vec{x})).$$

F is defined from G by expansion *if for all $\vec{f}$, $\vec{g}$, $\vec{x}$, and $\vec{y}$,*

$$F(\vec{f}, \vec{g}, \vec{x}, \vec{y}) = G(\vec{f}, \vec{x}).$$

F is defined from G, H, and K by limited recursion on notation (LRN) *if for all $\vec{f}$, $\vec{x}$, and y,*

$$\begin{aligned} F(\vec{f}, \vec{x}, 0) &= G(\vec{f}, \vec{x}), \\ F(\vec{f}, \vec{x}, y) &= H(\vec{f}, \vec{x}, y, F(\vec{f}, \vec{x}, \lfloor \tfrac{y}{2} \rfloor)), \qquad y > 0, \\ |F(\vec{f}, \vec{x}, y)| &\le |K(\vec{f}, \vec{x}, y)|. \end{aligned}$$

DEFINITION 3.2. *Let X be a set of type-2 functionals. The class of* basic feasible functionals defined from X (BFF(X)) *is the smallest class of functionals containing X, all type-1 poly-time functions and the application functional* Ap, *defined by* Ap$(f, x) = f(x)$, *and which is closed under functional composition, expansion, and limited recursion on notation. If $F \in BFF(X)$, we say that F is* basic feasible in X. *The* basic feasible functionals (BFFs) *are just $BFF(\emptyset)$.*[2] *If a functional F is in $BFF(\{F_1, \ldots, F_n\})$, we say that F is* feasible in $F_1, \ldots, F_n$.

The BFFs have a strong closure property with respect to computation by OTMs. We now define this property.

DEFINITION 3.3. *A class X of functionals has the* Ritchie–Cobham property *if for all F, $F \in X$ iff there is an OTM M and some $G \in X$ so that M computes F and for all inputs $\vec{f}$ and $\vec{x}$, the running time of M is bounded by $|G(\vec{f}, \vec{x})|$.*

The following result is due to Mehlhorn [9].

THEOREM 3.4. *The BFFs have the Ritchie–Cobham property.*

Mehlhorn proved this result for the model in which oracle calls have unit cost. It is not hard to adapt Mehlhorn's proof to prove the forward direction for our model. Note that simulating the application functional Ap requires more running time in our model because of the extra cost associated with an oracle call. However, this overhead can easily be incorporated into the time bound.

Sufficiency in our model follows from Mehlhorn's result. However, we need a somewhat stronger result in §5, which we now outline. We will show that for any OTM M, there is a BFF Run$_M$ such that the value of Run$_M(\vec{f}, \vec{x}, T)$, where $T \in \mathbb{N}$, is an encoding of a sequence of instantaneous descriptions (IDs) which give the history of the first Steps$(M, \vec{f}, \vec{x}, |T|)$ steps of M's execution with inputs $\vec{f}$ and $\vec{x}$. Below, we give an outline of how Run$_M$ is defined by limited recursion on notation (LRN) on T. Also, it is not hard to define a BFF Output so that if x codes a sequence of IDs, then Output(x) is the contents of the output tape of the last ID in the sequence. So if F is computed by an OTM M for which there is a BFF

[2] In [8], the definition of BFF(X) also includes closure under *functional substitution.* Results of Townsend [10] show that this definition is equivalent to the one given here.

G so that for all inputs $\vec{f}$ and $\vec{x}$, the running time of M is bounded by $|G(\vec{f}, \vec{x})|$, we have $F(\vec{f}, \vec{x}) = \text{Output}(\text{Run}_M(\vec{f}, \vec{x}, G(\vec{f}, \vec{x})))$, and so F is a BFF.

In order to define Run_M, we first use standard low-level encoding techniques, similar to those of [9], to define a BFF Next_M such that $\text{Next}_M(\vec{f}, \vec{x}, i)$ returns the ID which follows from ID i of M on inputs $\vec{f}$ and $\vec{x}$, assuming that i is a valid ID for M. In the case where the state associated with i is an oracle-query state, we use the Ap functional to obtain the resulting value. Now, to compute $\text{Run}_M(\vec{f}, \vec{x}, T)$, we use LRN on T to iterate Next_M $|T|$ times, starting with M's initial configuration. However, during each iteration, we also check that the overall running time (including the cost incurred for oracles calls) does not exceed $|T|$. If this is not the case, we "exit" from the iteration at this point. Finally, note that $|\text{Run}_M(\vec{f}, \vec{x}, T)|$ is $O(|T|(|\vec{x}| + |T|))$, independently of $\vec{f}$. Full details of this construction can be found in [7].

This result provides some evidence of the naturalness of the BFFs. However, it does not provide a purely machine-based characterization of type-2 feasibility. We will now introduce such a characterization.

4. Basic poly-time functionals. Recall that a type-1 function f is poly time if there is a TM M and a polynomial p such that M computes f and, for all inputs x, the running time of M with input x is bounded by $p(|x|)$. Hence a function is feasible if it is computable in time polynomial in the size of its input. A naive generalization of this characterization to type-2 functionals would lead us to propose that a functional is feasible if it is computable in time polynomial in the lengths of its inputs, where now its inputs include functions as well as numbers. In order to formalize this proposal, we need to answer two questions. The first is, what is the "length" of a function input f? Since f is an infinite object, there can be no single $n \in \mathbb{N}$ which measures the length of f. However, for each x there is an associated length $|f(x)|$, which is also the cost of querying f at x. Viewing f as a subroutine, there is a worst-case complexity for calling f, given this query cost. It is this complexity which we define to be the length of f.

DEFINITION 4.1. *For any $f \in^{\mathbb{N}} \mathbb{N}$, the* length *of f, denoted $|f|$, is the function defined by*

$$|f|(n) = \max_{|y| \leq n} |f(y)|.$$

Given this definition for the length of a function, we are presented with a second question: when is a functional "polynomial in" the length of a function? We answer this question by generalizing polynomials to allow function variables.

DEFINITION 4.2. First-order variables *are elements of the set* $\{n_1, n_2, \dots\}$. Second-order variables *are elements of the set* $\{L_1, L_2, \dots\}$. Second-order polynomials *are defined inductively: any $c \in \mathbb{N}$ is a second-order polynomial*; *first-order variables are second-order polynomials*; *and if P and Q are second-order polynomials and L is a second-order variable, then $P + Q$, $P \cdot Q$, and $L(P)$ are second-order polynomials.*

We will refer to second-order polynomials as polynomials when the context makes this distinction clear. Suppose P is a polynomial, all of whose first-order variables are among $n_1, \dots, n_s$ and all of whose second-order variables are among $L_1, \dots, L_t$. Then for any sequence $f_1, \dots, f_t$ of functions and any sequence $x_1, \dots, x_s$ of numbers, $P(\vec{f}, \vec{x})$, *P evaluated at $\vec{f}, \vec{x}$* denotes some natural number. For example, if

$$P_0 = L_1(L_1(n_1 \cdot n_1)) + L_1(L_1(n_1) \cdot L_1(n_1)) + L_1(n_1) + 4 \tag{1}$$

and $f(x) = x^2$, then

$$\begin{aligned} P_0(f,2) &= f(f(2\cdot 2)) + f(f(2)\cdot f(2)) + f(2) + 4 \\ &= (4^2)^2 + (2^2\cdot 2^2)^2 + 2^2 + 4 \\ &= 520. \end{aligned}$$

We are now ready to introduce a type-2 analogue for the poly-time functions, based on our generalizations of polynomials and lengths for functions.

DEFINITION 4.3. *A functional F is* basic poly time *if there is an OTM M and a second-order polynomial P such that M computes F, and for all $\vec{f}$ and $\vec{x}$, $T_M(\vec{f},\vec{x})$ is bounded by $P(|f_1|,\ldots,|f_k|,|x_1|,\ldots,|x_l|)$.*

Note that if M and P are as in the preceding definition, we will say that P *bounds the running time of M.*

5. Equivalence of basic feasible and basic poly-time functionals. We have proposed a new definition for type-2 feasibility, namely basic poly-time computability. In this section, we will show that our new definition coincides with Mehlhorn's. We begin with the following theorem.

THEOREM 5.1. *Every BFF is basic poly time.*

In order to prove the theorem, we require the following easily demonstrated facts.

LEMMA 5.2. *Suppose P and Q are polynomials with first-order variables $n_1,\ldots,n_l$ and second-order variables $L_1,\ldots,L_{k+1}$. Then for all i, $1\le i\le l$, there is a polynomial P' so that for all $\vec{g}$, $\vec{x}$,*

$$P'(\vec{g},\vec{x}) = P(\vec{g},x_1,\ldots,x_{i-1},Q(\vec{g},\vec{x}),x_{i+1},\ldots,x_l).$$

LEMMA 5.3. *Suppose P is a polynomial with first-order variables $n_1,\ldots,n_l$ and second-order variables $L_1,\ldots,L_k$. Then for all monotone nondecreasing $g_1,\ldots,g_k$ and all $x_1,\ldots,x_l$, and for all i, $1\le i\le l$, and all y, if $y\ge x_i$, then*

$$P(\vec{g},x_1,\ldots,x_{i-1},y,x_{i+1},\ldots,x_l)\ge P(\vec{g},\vec{x}).$$

Proof of Theorem 5.1. By Theorem 3.4, it suffices to show that if F is a BFF, then there is a polynomial P so that for all $\vec{f}$, $\vec{x}$, $|F(\vec{f},\vec{x})|\le P(|f_1|,\ldots,|f_k|,|x_1|,\ldots,|x_l|)$. We proceed by induction on the definition of F. The result is clear when F is an initial function. We now consider each definition scheme. In each case, we assume that F is defined from functionals for which the theorem holds. The case of expansion is straightforward. If F is defined from F, G, and K by LRN and P is a bounding polynomial for K, then it is also a bounding polynomial for F (since $|F(\vec{f},\vec{x})|\le|K(\vec{f},\vec{x})|$). Now suppose that F is defined from $H,G_1,\ldots,G_l$ by functional composition, and suppose that P is a bounding polynomial for H and P_i is a bounding polynomial for G_i, $1\le i\le l$. By Lemma 5.2, there is a polynomial P' such that for all $\vec{f}$ and $\vec{x}$,

$$\begin{aligned} &P'(|f_1|,\ldots,|f_k|,|x_1|,\ldots,|x_l|) \\ &= P(|f_1|,\ldots,|f_k|,P_1(|f_1|,\ldots,|f_k|,|x_1|,\ldots,|x_l|),\ldots, \\ &\quad P_l(|f_1|,\ldots,|f_k|,|x_1|,\ldots,|x_l|)), \end{aligned}$$

so by Lemma 5.3, $|G(\vec{f},\vec{x})|\le P'(|f_1|,\ldots,|f_k|,|x_1|,\ldots,|x_l|)$. □

Surprisingly, the converse of 5.1 is also true, so that the BFFs and the basic poly-time functionals coincide. We begin by considering an example which illustrates some of the problems associated with proving the converse.

Let F_1 be defined as follows:

$$F_1(f,x) = \begin{cases} (\mu k < x)(\max_{i \le k} |f(i)| = k) & \text{if such a } k \text{ exists,} \\ x & \text{otherwise.} \end{cases}$$

It is easy to see that this functional is basic poly time. For inputs f and x we can compute F_1 in time bounded by $c_1[|f|(|x|)]^2 + c_2$ for constants c_1 and c_2 as follows: just evaluate f at successive inputs, starting with 0, until we find a point k such that $F_1(f,x) = k$ or reach x. Now we will make at most $F_1(f,x) + 1$ such evaluations, and each evaluation returns a value with length bounded by $|f|(|x|)$. The approximate run-time bound is then obtained by noting that $F_1(f,x) \le |f|(|x|)$. This approach to computing F_1 does not allow us to conclude that F_1 is a BFF. In particular, it appears that with such an approach, computing F_1 for certain inputs f and x would require a recursion with an exponential number of iterations. However, this problem can be avoided with a nested recursion, as we will now show. In order to do so, we need to consider the auxiliary function F_2:

$$F_2(f,x) = \begin{cases} (\mu k \le F_1(f,x))(f(k) = \max_{0 \le i \le F_1(f,x)} f(i)) & \text{if such a } k \text{ exists,} \\ x & \text{otherwise.} \end{cases}$$

$F_2(f,x)$ returns the smallest point y, $0 \le y \le F_1(f,x)$, such that y maximizes f over $\{0, \ldots, F_1(f,x)\}$. So if $F_1(f,x) < x$, $|f(F_2(f,x))| = F_1(f,x)$. Otherwise, $|f(F_2(f,x))| \ge x$. Let

$$F(f,x) = \langle F_1(f,x), F_2(f,x) \rangle$$

and let

$$G(f,x,y) = \langle F_1(f, \min(|x|, y)), F_2(f, \min(|x|, y)) \rangle,$$

where $\langle \cdot, \cdot \rangle$ is a poly-time pairing function. Clearly G is basic feasible (we can use LRN on x to do a "brute-force" search). Let # be the rank (0, 2) BFF defined by $x\#y = 2^{|x|\cdot|y|}$. We define F using LRN, as follows:

$$\begin{cases} F(f,0) = \langle 0,0 \rangle, \\ F(f,x) = \textbf{if } \Pi_1(F(f, \lfloor \frac{x}{2} \rfloor)) < \lfloor \frac{x}{2} \rfloor \textbf{ then } F(f, \lfloor \frac{x}{2} \rfloor), \\ \qquad\qquad \textbf{else } G(f, f(\Pi_2(F(f, \lfloor \frac{x}{2} \rfloor)))\#2, x), \\ |F(f,x)| \le |\langle x,x \rangle|, \end{cases}$$

where Π_1 and Π_2 are poly-time projection functions. So $F_1(f,x) = \Pi_1(F(f,x))$.

To simplify the presentation of our result, we will restrict our attention to functionals of rank (1, 1). Basically, we want to show that if F is computed by an OTM M with running time bounded by P, then there is a BFF G so that for all f and x, the running time of M is bounded by $|G(f,x)|$. More formally, our goal is to find a BFF G so that $F(f,x) = \text{Output}(\text{Run}_M(f,x,G(f,x)))$. Now if there were a BFF H such that $|H(f,x)| \ge |f|(|x|)$, our task would be trivial, since we could then obtain the BFF G from H. It is not hard to show that there is no BFF H with the required property.

LEMMA 5.4. *For any BFF F, if f is a* 0–1*-valued function, then there is a polynomial p such that for all x, $|F(f,x)| \le p(|x|)$.*

Proof. Use a straightforward induction on the definition of F. □

THEOREM 5.5. *If H is a functional such that for all f and x, $|H(f, x)| \geq |f|(|x|)$, then H is not basic feasible.*

Proof. Assume that H is a BFF, and let f be a 0–1-valued function. By Theorem 3.4 and Lemma 5.4, there is an OTM M and a polynomial p such that for all x, $H(f, x)$ is computable by M in time $p(|x|)$. So in computing $H(f, x)$, M can query f at no more that $p(|x|)$ different inputs, and thus for some x_0 there is a y, $|y| \leq |x_0|$, so that M, on input f and x_0, does not query f at y. Let f' be defined by

$$f'(x) = \begin{cases} 2 \cdot H(f, x) + 1 & \text{if } x = y, \\ f(x) & \text{otherwise.} \end{cases}$$

Now by Proposition 2.3, M behaves identically with inputs f, x_0 and f', x_0, so that

$$|H(f, x_0)| = |H(f', x_0)| \geq |f'|(|x_0|) = |H(f, x_0)| + 1,$$

and we have derived a contradiction. □

Our goal now is to try to simplify P in such a way that the value of $P(|f|, |x|)$ can be feasibly computed without using a functional such as H. We begin by noting the following facts regarding running times.

LEMMA 5.6. *Suppose M is an OTM and P is a polynomial which bounds the running time of M. For any f, x, and t, if $Q = Q(M, f, x, t)$ and $t > P(|f_Q|, |x|)$, then $S_M(f_Q, x) <$ Steps(t).*

Proof. Suppose this is not the case for some t, and let $Q = Q(M, f, x, t)$. Since $S_M(f_Q, x) = \text{Steps}(T_M(f_Q, x))$ and Steps is monotone increasing, $T_M(f_Q, x) \geq t$. So $T_M(f_Q, x) > P(|f_Q|, |x|)$, contrary to Proposition 2.3. □

LEMMA 5.7. *Suppose M is an OTM and P is a polynomial which bounds the running time of M. For any f, x, t, and t', if $Q = Q(M, f, x, t)$, $Q' = Q(M, f, x, t')$, and $t' > P(|f_Q|, |x|)$, then either $|Q'| > |Q|$ or $S_M(f_{Q'}, x) <$ Steps(t').*

Proof. Suppose $|Q'| = |Q|$. Then $Q' = Q$, so $P(|f_{Q'}|, |x|) = P(|f_Q|, |x|)$. But then $t' > P(|f_{Q'}|, |x|)$, and so by Lemma 5.6, $S_M(f_{Q'}, x) < \text{Steps}(t')$. □

We denote by $t(M, f, x, r)$ the least value t such that $\text{Steps}(t) = S_M(f, x)$ or $|Q(M, f, x, t)| = r$. We will abbreviate this by $t(r)$ in appropriate contexts.

LEMMA 5.8. *Suppose $t = t(M, f, x, r)$ and $Q = Q(M, f, x, t)$ for $r \in \mathbb{N}$. If* Steps$(t) <$ *$S_M(f, x)$, then* Steps$(t) \leq$ *$S_M(f_Q, x)$.*

Proof. Given that $\text{Steps}(t) < S_M(f, x)$, M with input f and x must query f at r distinct inputs before halting. But then by Proposition 2.3, the same is true of M with inputs f_Q, x. The result follows by the minimality of t. □

We now want to show that for our bounding polynomial P, there is a $d \in \mathbb{N}$ and a first-order polynomial $\hat{P}$ so that for any query set Q, there are points $q_1, \ldots, q_d$ in Q such that

$$P(|f_Q|, |x|) \leq \hat{P}(|f(q_1)|, \ldots, |f(q_d)|, |x|).$$

This will reduce our problem of finding a basic feasible bounding function for M to the problem of finding BFFs which gives us such $q_1, \ldots, q_d$ in $Q(M, f, x, P(|f|, |x|))$. We now describe the method for obtaining the first-order polynomial $\hat{P}$.

DEFINITION 5.9. *Let P be a polynomial. We define $d(P)$, the* depth *of P, by induction on P: $d(c) = d(n_i) = 0$; $d(P+Q) = d(P \cdot Q) = \max\{d(P), d(Q)\}$; and $d(L(P)) = 1+d(P)$.*

Let P be a polynomial with depth d. For $1 \leq c \leq d$, let $P^c_1, \ldots, P^c_{k_c}$ be an enumeration of the depth-c subpolynomials of P which have the form $L(Q)$, where L is a second-order variable and $d(Q) = c - 1$. If $P_j^c = L(Q)$, let $Q_j^c = Q$. Clearly, for any Q_j^c there is a first-order polynomial $\hat{Q}_j^c$, $1 \leq i < c$, $1 \leq l \leq k_c$, such that for all $\vec{g}$ and $\vec{x}$,

$$Q_j^c(\vec{g}, \vec{x}) = \hat{Q}_j^c(P_1^1(\vec{g}, \vec{x}), \ldots, P_{k_1}^1(\vec{g}, \vec{x}), \ldots, P_{k_{c-1}}^{c-1}(\vec{g}, \vec{x}), \vec{x})).$$

Now suppose that M is an OTM and that P is a depth-d polynomial bounding the running time of M. For any inputs f and x and any t, there are $q_1, \ldots, q_d \in \mathcal{Q}(M, f, x, t)$ so that for $1 \leq c \leq d$ and $1 \leq j \leq k_c$,

$$|q_c| \leq \max_{1 \leq j \leq k_c} Q_j^c(|f_{\mathcal{Q}}|, |x|) \tag{2}$$

and

$$|f(q_c)| \geq \max_{1 \leq j \leq k_c} P_j^c(|f_{\mathcal{Q}}|, |x|). \tag{3}$$

But then there are first-order polynomials $\hat{Q}_c$, $1 \leq c \leq d$ such that for $1 \leq j \leq k_c$,

$$Q_j^c(|f_{\mathcal{Q}}|, |x|) \leq \hat{Q}_c(|f(q_1)|, \ldots, |f(q_{c-1})|, |x|). \tag{4}$$

Similarly, there is a first-order polynomial $\hat{P}$ such that

$$P(|f_{\mathcal{Q}}|, |x|) \leq \hat{P}(|f(q_1)|, \ldots, |f(q_d)|, |x|).$$

As an example, for the polynomial P_0 given in (1), we have

$$\begin{aligned} P_0(|f_{\mathcal{Q}}|, |x|) &\leq |f_{\mathcal{Q}}|(|f_{\mathcal{Q}}|(|x|^2)) + |f_{\mathcal{Q}}|([|f_{\mathcal{Q}}|(|x|)]^2) + |f_{\mathcal{Q}}|(|x|) + 4 \\ &\leq |f_{\mathcal{Q}}|(|f(q_1)|) + |f_{\mathcal{Q}}|(|f(q_1)|^2) + |f(q_1)| + 4 \\ &\leq |f(q_2)| + |f(q_2)| + |f(q_1)| + 4, \end{aligned}$$

where $|q_1| \leq |x|^2$ and $|q_2| \leq |f(q_1)|^2$. So we have $\hat{Q}_1 = n_1^2$, $\hat{Q}_2 = n_1^2$, and $\hat{P} = 2 \cdot n_2 + n_1 + 4$.

DEFINITION 5.10. *Let M be an OTM whose running time is bounded by the depth-d polynomial P. For* $1 \leq c \leq d$, *the cth* maximizing argument *for M, f, x, and t is the least-value q_c satisfying* (2) *and* (3) *for $\mathcal{Q} = \mathcal{Q}(M, f, x, t)$. The cth maximizing argument for M, f, x, and t is denoted $q_c(M, f, x, t)$, or just $q_c(t)$ when M, f, and x are understood.*

Recall that for any first-order polynomial $p(n_1, \ldots, n_k)$ with positive coefficients, there is a poly-time function f_p so that $|f_p(x_1, \ldots, x_k)| = p(|x_1|, \ldots, |x_k|)$, for all $x_1, \ldots, x_k$. Since the application functional is a BFF, for $\hat{Q}_1, \ldots, \hat{Q}_d, \hat{P}$, there are BFFs $G_{\hat{Q}_1}, \ldots, G_{\hat{Q}_d}, G_{\hat{P}}$ so that for all $f, x, q_1, \ldots, q_d$,

$$|G_{\hat{Q}_c}(f, q_1, \ldots, q_{c-1}, x)| = \hat{Q}_c(|f(q_1)|, \ldots, |f(q_{c-1})|, |x|), \quad 1 \leq c \leq d,$$

and

$$|G_{\hat{P}}(f, q_1, \ldots, q_d, x)| = \hat{P}(|f(q_1)|, \ldots, |f(q_d)|, |x|).$$

Now suppose that there are BFFs $G_1, \ldots, G_d$ such that for $1 \leq c \leq d$,

$$|f(G_c(f, x))| \geq |f(q_c(M, f, x, P(|f|, |x|))|.$$

We could then define a basic feasible bounding functional G by

$$G(f, x) = G_{\hat{P}}(f, G_1(f, x), \ldots, G_d(f, x), x).$$

As a first step toward finding such functionals, we will introduce a parameter r, which bounds the number of inputs at we allow M to query f. Formally, if $d(P) = d$, let $\mathrm{Max\_Arg}^c_M$, $1 \le c \le d$, be the functional of rank $(1, 2)$ defined by

$$\mathrm{Max\_Arg}^c_M(f, x, r) = \begin{cases} 0 & \text{if } r = 0, \\ q_c(M, f, x, t) + 1 & \text{otherwise,} \end{cases}$$

where t is the smallest value such that $S_M(f, x) \le \mathrm{Steps}(t)$ or $|Q(M, f, x, t)| \ge r$. In general, we will denote such a t by $t(M, f, x, r)$, or just $t(r)$ in the appropriate context.

CLAIM 5.11. *For any OTM M with running time bounded by a polynomial P with depth d,* $\mathrm{Max\_Arg}^c_M$ *is a BFF for* $1 \le c \le d$.

Given this claim, we only need to show that we can feasibly eliminate the use of the parameter r. If we could show that there is a BFF R such that for all f and x, $R(f, x) = |Q(M, f, x, P(|f|, |x|))|$, we would be finished, because we would then have

$$\mathrm{Max\_Arg}^c_M(f, x, R(f, x)) - 1 = q_c(M, f, x, P(|f|, |x|)).$$

What we will actually show is that there is a constant d and a sequence $r_1, \ldots, r_d$ of "approximations" to $R(f, x)$ such that r_1 is basic feasible in f, x and r_{c+1} is basic feasible in r_c, f, and x, and such that a basic feasible bounding functional G can be obtained from the r_c's. Intuitively, r_c is an upper bound on $R(f, x)$, assuming that M queries f only at points y such that $|y| \le |q_c|$. Given r, let

$$(5) \qquad T = G_{\hat{P}}(f, \mathrm{Max\_Arg}^1_M(f, x, r) - 1, \ldots, \mathrm{Max\_Arg}^d_M(f, x, r) - 1, x),$$

and $Q = Q(M, f, x, t)$, where $t = t(r)$. If $S_M(f, x) \le \mathrm{Steps}(t)$, $|T| \ge P(|f|, |x|)$, and so $F(f, x) = \mathrm{Output}(\mathrm{Run}_M(f, x, T))$. Otherwise, $S_M(f_Q, x) \ge \mathrm{Steps}(t)$ by Lemma 5.8, and so, by Lemma 5.6, $P(|f_Q|, |x|) \ge t$. By the definition of T, $|T| \ge P(|f_Q|, |x|)$. Finally, since each query made by M has at least unit cost, $t \ge r$. Combining these inequalities gives $|T| \ge r$. Now there is a BFF A which satisfies

$$|f(A(f, x))| = \max_{y \le |x|} |f(y)|.$$

Since $|T| \ge r$, $|f(A(f, 2\#T))| \ge |f|(|r|)$. In other words, if M on inputs f and x runs for long enough to query f at r inputs, then we can feasibly compute an upper bound of $|f|(|r|)$ from f, x, and r.

We will now give an example to show how we take advantage of the approach described above. Recall the polynomial P_0 given in (1).

$$P_0(|f|, |x|) = |f|(|f|(|x|^2)) + |f|([|f|(|x|)]^2) + |f|(|x|) + 4.$$

Suppose P_0 bounds the running time of M which computes the functional F. For inputs f and x, we will begin by trying to find q_1. For any Q, $|q_1| \le |x|^2 \le |x\#x|$. So we begin by setting $r_1 = x\#x$. Let T_1 be T as defined in (5), for $r = r_1$. Now if M halts before making r_1 queries, it halts in $\mathrm{Steps}(|T_1|)$ steps, so we do not need to go any further, since $|T_1|$ will bound the running time of M. Otherwise, we have a value $l_1 = A(f, 2\#T_1)$ so that $|f(l_1)| \ge |f(q_1)|$. Since $|q_2| \le |f(q_1)|^2$, we now try $r_2 = f(l_1)\#f(l_1)$, and let T_2 be T as defined by (5) for $r = r_2$. Again, if M halts in $\mathrm{Steps}(|T_2|)$ steps, we are done. Otherwise, we have a value $l_2 = A(f, 2\#T_2)$ so that $|f(l_2)| \ge |f(q_2)|$. Under the assumption that for inputs f and x, M does not halt in $\mathrm{Steps}(|T_1|)$ or $\mathrm{Steps}(|T_2|)$ steps, the running time of M must be bounded by $2 \cdot |f(l_2)| + |f(l_1)| + 4$. So if Pad is the rank $(0, 2)$ BFF defined by $\mathrm{Pad}(x, y) = x \cdot 2^{|y|}$ and

$$G(f, x) = \max\{T_1, T_2, \mathrm{Pad}(\mathrm{Pad}(2\#\mathrm{Ap}(f, l_2), \mathrm{Ap}(f, l_1)), 16)\},$$

then

$$F(f, x) = \text{Output}(\text{Run}_M(f, x, G(f, x))).$$

Formalizing this argument for arbitrary bounding polynomials, we obtain our main result.

THEOREM 5.12. *If F is a rank-$(1, 1)$ basic poly-time functional, then F is basic feasible.*

Proof. Suppose that F is a computed by an OTM M such that for all f and x, the running time of M with inputs f, x is bounded by $P(|f|, |x|)$, where $d(P) = d$. Let

$$\begin{aligned}
r_1 &= G_{\hat{Q}_1}(f, x), \\
T_c &= G_{\hat{P}}(f, \text{Max\_Arg}^1_M(f, x, r_c) - 1, \ldots, \text{Max\_Arg}^d_M(f, x, r_c) - 1, x), && 1 \le c \le d, \\
l_c &= A(f, 2\#T_c), && 1 \le c \le d, \\
r_c &= G_{\hat{Q}_c}(f, l_1, \ldots, l_{c-1}, x), && 2 \le c \le d,
\end{aligned}$$

and define G by

$$G(f, x) = \max\{G_{\hat{P}}(f, l_1, \ldots, l_d, x), T_1, \ldots, T_d\}.$$

An argument similar to that given in the example shows that for all f and x,

$$F(f, x) = \text{Output}(\text{Run}_M(f, x, G(f, x))). \qquad \square$$

It remains to demonstrate Claim 5.11. We want to show that Max_Arg^c_M is a BFF for $1 \le c \le d$. To do so, we also consider for $1 \le c \le d$, the functional Max_Arg_Unary^c_M of rank $(1, 2)$ such that for all f, x, r, and R, if $|R| = r$ then

$$\text{Max\_Arg}^c_M(f, x, r) = \text{Max\_Arg\_Unary}^c_M(f, x, R).$$

We begin by showing that Max_Arg_Unary^c_M is feasible for case where $d = 1$.

LEMMA 5.13. *For any OTM M with running time bounded by a depth-1 polynomial P, Max_Arg_Unary^1_M is basic feasible.*

Proof. Max_Arg_Unary^1_M is defined by LRN on R. $\text{Max\_Arg\_Unary}^1_M(f, x, 0) = 0$. Now suppose that we have defined $\text{Max\_Arg\_Unary}^1_M(f, x, \lfloor \frac{R}{2} \rfloor)$. Let $t = t(|\lfloor \frac{R}{2} \rfloor|)$, $Q = Q(t)$, and

$$T = 2 \cdot G_{\hat{P}}(f, \text{Max\_Arg\_Unary}^1_M(f, x, \lfloor \tfrac{R}{2} \rfloor) - 1, x) + 1.$$

Finally, let $h = \text{Run}_M(f, x, T)$. We claim that h encodes enough of the computation of M so that $\text{Max\_Arg\_Unary}(f, x, R)$ can be feasibly computed from h. If $S_M(f, x) \le \text{Steps}(t)$, then $|T| \ge P(|f|, |x|)$, so the claim holds. Otherwise, by Lemma 5.8, $S_M(f_Q, x) \ge \text{Steps}(t)$. Since T is defined so that $|T| > P(|f_Q|, |x|)$, we have, by Lemma 5.7, that $|Q(M, f, x, |T|)| > |Q(M, f, x, t)|$. Since $t = t(|\lfloor \frac{R}{2} \rfloor|)$, it follows that $|Q(M, f, x, |T|)| > |R|$, and again the claim holds. Since $|\text{Max\_Arg\_Unary}^1_M(f, x, R)|$ is bounded by $|G_{\hat{Q}_1}(f, x)|$, we conclude that Max_Arg_Unary^1_M is basic feasible. $\square$

LEMMA 5.14. *For any M with running time bounded by a depth-1 polynomial P, Max_Arg^1_M is basic feasible.*

Proof. Max_Arg^1_M is defined by LRN on r. We set $\text{Max\_Arg}^1_M(f, x, 0) = 0$. Now suppose that we have defined $\text{Max\_Arg}^1_M(f, x, \lfloor \frac{r}{2} \rfloor)$. Let Let $t = t(\lfloor \frac{r}{2} \rfloor)$, $Q = Q(t)$, and

$$T = 2 \cdot G_{\hat{P}}(f, \text{Max\_Arg}^1_M(f, x, \lfloor \tfrac{r}{2} \rfloor) - 1, x) + 1.$$

Finally, let $h = \text{Run}_M(f, x, T)$. If $S_M(f, x) \leq \text{Steps}(t)$, $|T| > P(|f|, |x|)$, and so h encodes the whole history of M's execution on inputs f, x and we can obtain $\text{Max\_Arg}^1_M(f, x, r)$ via a feasible computation on this history.

Otherwise, $S_M(f_Q, x) \geq \text{Steps}(t)$ by Lemma 5.8, so by Lemma 5.6, $P(|f_Q|, |x|) \geq t$. Now T is defined so that $|T| > P(|f_Q|, |x|)$, so $|T| > t$. Since every query has at least unit cost, $t \geq \lfloor \frac{r}{2} \rfloor$, and so we have $|T| \geq r$. It is easy to show that there is a poly-time function $s(x, y)$ so that if $|x| > y$, then $|s(x, y)| = y$. So

$$\text{Max\_Arg}^1_M(f, x, r) = \text{Max\_Arg\_Unary}^1_M(f, x, s(T, r)).$$

Since

$$|\text{Max\_Arg}^1_M(f, x, R)| \leq |G_{\hat{Q}_1}(f, x)|,$$

we conclude that Max_Arg^1_M is basic feasible. □

It is not hard to see that this argument can be extended to give a simultaneous definition of $\text{Max\_Arg\_Unary}^c_M(f, x, r)$ and $\text{Max\_Arg}^c_M(f, x, y)$, $1 \leq c \leq d$, assuming that the running time of M is bounded by a depth-d polynomial. So before we can conclude that these functions are BFFs as in Claim 5.11, we must extend LRN to allow simultaneous definitions. We begin by considering a simple extension of LRN.

Definition 5.15. *$F_1, \ldots, F_k$ are defined from $\vec{G}$, $\vec{H}$, and $\vec{K}$ by* simultaneous limited recursion on notation (SLRN) *iff for all $\vec{f}$, $\vec{x}$, and y,*

$$F_i(\vec{f}, \vec{x}, 0) = G_i(\vec{f}, \vec{x}), \quad 1 \leq i \leq k,$$

$$F_i(\vec{f}, \vec{x}, y) = H_i(\vec{f}, \vec{x}, y, \vec{F}(\vec{f}, \vec{x}, \lfloor \tfrac{y}{2} \rfloor)), \quad y > 0, 1 \leq i \leq k,$$

$$|F_i(\vec{f}, \vec{x}, y)| \leq |K_i(\vec{f}, \vec{x}, y)|, \quad 1 \leq i \leq k.$$

Lemma 5.16. *If $F_1, \ldots, F_k$ are defined from $\vec{G}$, $\vec{H}$, and $\vec{K}$ by SLRN, then F_i is basic feasible in $\vec{G}$, $\vec{H}$, and $\vec{K}$, $1 \leq i \leq k$.*

Proof. Recall that for $k \in \mathbb{N}$, there are poly-time functions $\lambda x_1 \ldots \lambda x_k . \langle x_1, \ldots, x_k \rangle$ and Π^k_i, $1 \leq i \leq k$, such that $\Pi^k_i(\langle x_1, \ldots, x_k \rangle) = x_i$. Now it is not hard to see that we can define, using LRN, a functional F, basic feasible in $\vec{G}$, $\vec{H}$, and $\vec{K}$, such that $F_i(\vec{f}, \vec{x}, y) = \Pi^k_i(F(\vec{f}, \vec{x}, y))$. □

It would appear that SLRN is too weak to allow the simultaneous definition of the functionals Max_Arg^c_M, $1 \leq c \leq d$. The problem arises in attempting to provide bounds for these functionals. By (2) and (4), we know that

$$|\text{Max\_Arg}^c_M(\vec{f}, \vec{x}, y)| \leq |G_{\hat{Q}_c}(f, \text{Max\_Arg}^1_M(\vec{f}, \vec{x}, y) - 1, \ldots, \text{Max\_Arg}^{c-1}_M(\vec{f}, \vec{x}, y) - 1, x)|$$

and $\text{Max\_Arg\_Unary}^c_M(\vec{f}, \vec{x}, r)$ is similarly bounded. However, the bounding conditions for SLRN do not allow such a general form of bounding. We now introduce a form of simultaneous recursion which does.

Definition 5.17. *$F_1, \ldots, F_k$ are defined from $\vec{G}$, $\vec{H}$, and $\vec{K}$ by* multiple limited recursion on notation (MLRN) *iff for all $\vec{f}$, $\vec{x}$, and y,*

$$F_i(\vec{f}, \vec{x}, 0) = G_i(\vec{f}, \vec{x}), \quad 1 \leq i \leq k,$$

$$F_i(\vec{f}, \vec{x}, y) = H_i(\vec{f}, \vec{x}, y, \vec{F}(\vec{f}, \vec{x}, \lfloor \tfrac{y}{2} \rfloor)), \quad y > 0, 1 \leq i \leq k,$$

$$|F_1(\vec{f}, \vec{x}, y)| \leq |K_1(\vec{f}, \vec{x}, y)|,$$

$$|F_i(\vec{f}, \vec{x}, y)| \leq |K_i(\vec{f}, \vec{x}, y, F_1(\vec{f}, \vec{x}, y), \ldots, F_{i-1}(\vec{f}, \vec{x}, y))|, \quad 2 \leq i \leq k.$$

MLRN is a generalization of a scheme introduced by Cook in [2]. The apparent power of this scheme compared to SLRN arises from the use of weaker bounds for $|F_i(\vec{f}, \vec{x}, y)|$, which require bounding only in terms of $F_1, \ldots, F_{i-1}$. Otherwise, MLRN is identical to SLRN. Surprisingly, it is no more powerful than SLRN.

THEOREM 5.18. *If $F_1, \ldots, F_k$ are defined by MLRN from $\vec{G}$, $\vec{H}$, and $\vec{K}$, then F_i is basic feasible in $\vec{G}$, $\vec{H}$, and $\vec{K}$, $1 \leq i \leq k$.*

This is a result which is interesting in its own right. We postpone its proof to the following section, and continue now with the thread of our main result.

Proof of Claim 5.11. We extend the proof of Lemma 5.14, to allow a simultaneous definition of Max_Arg_Unary^c_M, $1 \leq c \leq d$, using MLRN. We then use these functionals to give a simultaneous definition of Max_Arg^c_M, $1 \leq c \leq d$, again using MLRN. □

6. Feasibility of MLRN. This section is devoted to the proof of Theorem 5.18. Our first step is to consider an apparently weaker version of MLRN.

DEFINITION 6.1. *$F_1, \ldots, F_k$ are defined from $\vec{G}$, $\vec{H}$, and $\vec{K}$ by* weak multiple limited recursion on notation (WMLRN) *iff for all $\vec{f}$, $\vec{x}$, and y,*

$$F_i(\vec{f}, \vec{x}, 0) = G_i(\vec{f}, \vec{x}), \quad 1 \leq i \leq k,$$

$$F_i(\vec{f}, \vec{x}, y) = H_i(\vec{f}, \vec{x}, y, F_1(\vec{f}, \vec{x}, \lfloor \tfrac{y}{2} \rfloor), \ldots, F_k(\vec{f}, \vec{x}, \lfloor \tfrac{y}{2} \rfloor)), \quad y > 0, 1 \leq i \leq k,$$

$$|F_1(\vec{f}, \vec{x}, y)| \leq |K_1(\vec{f}, \vec{x}, y)|,$$

$$|F_i(\vec{f}, \vec{x}, y)| \leq |K_i(\vec{f}, \vec{x}, y, F_{i-1}(\vec{f}, \vec{x}, y))|, \quad 2 \leq i \leq k.$$

Once we show that the BFFs are closed under WMLRN, we obtain a proof of Theorem 5.18, as follows.

Proof of Theorem 5.18. We show that there are functionals $F'_1, \ldots, F'_k$, basic feasible in $\vec{G}$, $\vec{H}$, and $\vec{K}$, so that for $1 \leq i \leq k$,

$$F'_i(\vec{f}, \vec{x}, y) = \langle F_1(\vec{f}, \vec{x}, y), \ldots, F_i(\vec{f}, \vec{x}, y) \rangle.$$

$F'_i, \ldots, F'_k$ are defined by WMRLN, as follows:

$$F'_i(\vec{f}, \vec{x}, 0) = \langle G_1(\vec{f}, \vec{x}), \ldots, G_i(\vec{f}, \vec{x}) \rangle,$$

$$F'_i(\vec{f}, \vec{x}, y) = \langle H_1(\vec{f}, \vec{x}, y, \Pi^k_1((F'_1(\vec{f}, \vec{x}, \lfloor \tfrac{y}{2} \rfloor))), \ldots, \Pi^k_k((F'_k(\vec{f}, \vec{x}, \lfloor \tfrac{y}{2} \rfloor)))), \ldots,$$

$$H_i(\vec{f}, \vec{x}, y, \Pi^k_1((F'_1(\vec{f}, \vec{x}, \lfloor \tfrac{y}{2} \rfloor))), \ldots, \Pi^k_k((F'_k(\vec{f}, \vec{x}, \lfloor \tfrac{y}{2} \rfloor))))) \rangle, \quad y > 0,$$

$$|F'_1(\vec{f}, \vec{x}, y)| \leq |K_1(\vec{f}, \vec{x}, y)|,$$

$$|F'_i(\vec{f}, \vec{x}, y)| \leq |\langle K_1(\vec{f}, \vec{x}, y), K_2(\vec{f}, \vec{x}, y, \Pi^{i-1}_1(F'_{i-1}(\vec{f}, \vec{x}, y))), \ldots,$$

$$K_i(\vec{f}, \vec{x}, y, \Pi^{i-1}_1(F'_{i-1}(\vec{f}, \vec{x}, y)), \ldots, \Pi^{i-1}_{i-1}(F'_{i-1}(\vec{f}, \vec{x}, y))) \rangle| \qquad \square$$

It remains to show that the BFFs are closed under WMLRN. Suppose $F_1, \ldots, F_k$ are defined by WMLRN. By introducing extra parameters $z_1, \ldots, z_k$ so that F_i is bounded in terms of z_{i-1} rather than F_{i-1}, it is possible to define by SLRN functionals $E_1, \ldots, E_k$ so that E_i agrees with F_i when supplied with appropriate values for $z_1, \ldots, z_{k-1}$. This is formalized in Lemma 6.2 below. It then remains to define $z_1, \ldots, z_{k-1}$ with the required property. In fact, functionals giving appropriate values for $z_1, \ldots, z_{k-1}$ can be defined by WMLRN. Then, by induction on k, we can conclude that the F_i's are definable using SLRN. This is formalized in Lemma 6.3 below. We will write $x \subseteq y$ if for some i, $x = \lfloor y/2^i \rfloor$.

LEMMA 6.2. *Suppose that $F_1, \ldots, F_k$ are defined from $\vec{G}$, $\vec{H}$, and $\vec{K}$ by WMLRN. Then there are functionals $E_1, \ldots, E_k$, basic feasible in $\vec{G}$, $\vec{H}$, and $\vec{K}$, so that for all $z_1, \ldots, z_{k-1}$, $y'_1, \ldots, y'_{k-1}$ and all y, if*

$$(6) \qquad |F_i(\vec{f}, \vec{x}, v)| \leq |K_i(\vec{f}, \vec{x}, y'_{i-1}, z_{i-1})|, \quad 2 \leq i \leq k,$$

for all $v \subseteq y$, then

$$E_i(\vec{f}, \vec{x}, \vec{y'}, \vec{z}, y) = F_i(\vec{f}, \vec{x}, y), \quad 1 \leq i \leq k.$$

Proof. Let $L_1, \ldots, L_k$ be defined as follows:

$$L_1(\vec{f}, \vec{x}, \vec{y'}, \vec{z}, y) = K_1(\vec{f}, \vec{x}, y),$$

$$L_i(\vec{f}, \vec{x}, \vec{y'}, \vec{z}, y) = K_i(\vec{f}, \vec{x}, y'_{i-1}, z_{i-1}), \quad 2 \leq i \leq k.$$

Let $H'_1, \ldots, H'_k$ be defined as follows:

$$H'_i(\vec{f}, \vec{x}, \vec{y'}, \vec{z}, y, \vec{t}) = \begin{cases} H_i(\vec{f}, \vec{x}, y, \vec{t}) & \text{if } |H_i(\vec{f}, \vec{x}, y, \vec{t})| \leq |L_i(\vec{f}, \vec{x}, \vec{y'}, \vec{z}, y))|, \\ L_i(\vec{f}, \vec{x}, \vec{y'}, \vec{z}, y) & \text{otherwise.} \end{cases}$$

Finally, let $G'_1, \ldots, G'_k$ be defined as follows:

$$G'_i(\vec{f}, \vec{x}, \vec{y'}, \vec{z}) = G_i(\vec{f}, \vec{x}).$$

$E_1, \ldots, E_k$ are defined by SLRN from $\vec{G'}$, $\vec{H'}$, and $\vec{L}$. Clearly, E_i is bounded by L_i, $1 \leq i \leq k$. We now show by induction on y that if (6) holds for all $v \subseteq y$, then

$$E_i(\vec{f}, \vec{x}, \vec{y'}, \vec{z}, y) = F_i(\vec{f}, \vec{x}, y), \quad 1 \leq i \leq k.$$

This follows directly when $y = 0$. Now suppose it is the case for $\lfloor \frac{y}{2} \rfloor$. Assume that (6) holds for all $v \subseteq y$. Then it holds for all $v \subseteq \lfloor \frac{y}{2} \rfloor$, and by the induction hypothesis we have

$$E_i(\vec{f}, \vec{x}, \vec{y'}, \vec{z}, \lfloor \tfrac{y}{2} \rfloor) = F_i(\vec{f}, \vec{x}, \lfloor \tfrac{y}{2} \rfloor), \quad 1 \leq i \leq k,$$

and so

$$H_i(\vec{f}, \vec{x}, y, E_1(\vec{f}, \vec{x}, \vec{y'}, \vec{z}, \lfloor \tfrac{y}{2} \rfloor), \ldots, E_k(\vec{f}, \vec{x}, \vec{y'}, \vec{z}, \lfloor \tfrac{y}{2} \rfloor)) = F_i(\vec{f}, \vec{x}, y), \quad 1 \leq i \leq k.$$

Now $|F_1(\vec{f}, \vec{x}, y)| \leq |K_1(\vec{f}, \vec{x}, y)|$ and by our assumption that (6) holds for all $v \subseteq y$, for $2 \leq i \leq k$ we have

$$|F_i(\vec{f}, \vec{x}, y)| \leq |L_i(\vec{f}, \vec{x}, \vec{y'}, \vec{z}, y)|.$$

We then conclude, referring to (6), that $E_i(\vec{f}, \vec{x}, \vec{y'}, \vec{z}, y) = F_i(\vec{f}, \vec{x}, y)$. □

LEMMA 6.3. *If $F_1, \dots, F_k$ are defined by WMLRN from $\vec{G}$, $\vec{H}$, and $\vec{K}$, then F_i is basic feasible in $\vec{G}$, $\vec{H}$, and $\vec{K}$, $1 \leq i \leq k$.*

Proof. We proceed by induction on $k \geq 2$. When $k = 2$, we show that there is a functional P, basic feasible in $\vec{G}$, $\vec{H}$, $\vec{K}$ so that for all y and all $v \subseteq y$,

$$(7) \qquad |F_2(\vec{f}, \vec{x}, v)| \leq |K_2(\vec{f}, \vec{x}, \Pi_1(P(\vec{f}, \vec{x}, y)), \Pi_2(P(\vec{f}, \vec{x}, y)))|.$$

Having defined such a P, we can conclude from the preceding lemma that

$$F_i(\vec{f}, \vec{x}, y) = E_i(\vec{f}, \vec{x}, \Pi_1(P(\vec{f}, \vec{x}, y)), \Pi_2(P(\vec{f}, \vec{x}, y)), y), \quad i = 1, 2,$$

so that F_i is basic feasible in $\vec{G}$, $\vec{H}$, and $\vec{K}$, $i = 1, 2$. It is easy to see that if P satisfies

$$(8) \qquad P(\vec{f}, \vec{x}, y) = \langle v, F_1(\vec{f}, \vec{x}, v)\rangle,$$

where $v \subseteq y$ maximizes $|K_2(\vec{f}, \vec{x}, v, F_1(\vec{f}, \vec{x}, v))|$, then P will satisfy (7) for all $v \subseteq y$. So it suffices to find a BFF P which satisfies (8) for all y. Now define P as follows:

$$P(\vec{f}, \vec{x}, 0) = \langle 0, G_1(\vec{f}, \vec{x})\rangle,$$

$$P(\vec{f}, \vec{x}, y) = \begin{cases} \langle y, z_1\rangle & \text{if } |K_2(\vec{f}, \vec{x}, y, z_1)| \geq |K_2(\vec{f}, \vec{x}, \Pi_1(t_1), \Pi_2(t_1)))|, \\ t_1 & \text{otherwise,} \end{cases} \qquad y > 0,$$

where

$$z_1 = H_1(\vec{f}, \vec{x}, y, E_1(\vec{f}, \vec{x}, \Pi_1(t_1), \Pi_2(t_1), \lfloor \tfrac{y}{2} \rfloor), E_2(\vec{f}, \vec{x}, \Pi_1(t_1), \Pi_2(t_1), \lfloor \tfrac{y}{2} \rfloor))$$

and

$$t_1 = P(\vec{f}, \vec{x}, \lfloor \tfrac{y}{2} \rfloor).$$

We will show that P satisfies (8) for all y, by induction on the notation of y. This follows directly when $y = 0$. Now assume that P satisfies (8) for $\lfloor \frac{y}{2} \rfloor$. Then P satisfies (7) for all $v \subseteq \lfloor \frac{y}{2} \rfloor$, so that by the preceding lemma, $z_1 = F_1(\vec{f}, \vec{x}, y)$. But then, by the induction hypothesis and the definition of P,

$$P(\vec{f}, \vec{x}, y) = \begin{cases} \langle y, F_1(\vec{f}, \vec{x}, y)\rangle & \text{if } |K_2(\vec{f}, \vec{x}, y, F_1(\vec{f}, \vec{x}, y))| \geq |K_2(\vec{f}, \vec{x}, v, F_1(\vec{f}, \vec{x}, v))|, \\ \langle v, F_1(\vec{f}, \vec{x}, v)\rangle & \text{otherwise,} \end{cases}$$

where $v \subseteq \lfloor \frac{y}{2} \rfloor$ maximizes $|K_2(\vec{f}, \vec{x}, v, F_1(\vec{f}, \vec{x}, v))|$. It is then clear that P satisfies (8) for y, as required. Finally, since P satisfies (8) for all y,

$$|P(\vec{f}, \vec{x}, y)| \leq |\langle y, \max_{v \subseteq y} K_1(\vec{f}, \vec{x}, v)\rangle|,$$

so that, in fact, P is definable by LRN from functionals basic feasible in $\vec{G}$, $\vec{H}$, and $\vec{K}$.

Now assume that the result holds for $k - 1$. We show validity for k as follows: we will show that there are $P_1, \dots, P_{k-1}$, basic feasible in $\vec{G}$, $\vec{H}$, and $\vec{K}$ so that for $2 \leq i \leq k$,

$$(9) \qquad |F_i(\vec{f}, \vec{x}, v)| \leq |K_i(\vec{f}, \vec{x}, \Pi_1(P_{i-1}(\vec{f}, \vec{x}, y)), \Pi_2(P_{i-1}(\vec{f}, \vec{x}, y)))|$$

for all y and all $v \subseteq y$. By the preceding lemma, we will then have

$$F_i(\vec{f}, \vec{x}, y) = E_i(\vec{f}, \vec{x}, \Pi_1(P_1(\vec{f}, \vec{x}, y)), \dots, \Pi_1(P_{i-1}(\vec{f}, \vec{x}, y)), \Pi_2(P_1(\vec{f}, \vec{x}, y)), \dots,$$
$$\Pi_2(P_{i-1}(\vec{f}, \vec{x}, y), y)),$$

so that F_i is basic feasible in $\vec{G}$, $\vec{H}$, and $\vec{K}$, $1 \leq i \leq k$. We will attempt to define $P_1, \ldots, P_{k-1}$ by WMLRN. By the induction hypothesis, we can then conclude that $P_1, \ldots, P_{k-1}$ are basic feasible in $\vec{G}$, $\vec{H}$, and $\vec{K}$. It is easy to see that for $1 \leq i \leq k-1$, if P_i satisfies

$$(10) \qquad P_i(\vec{f}, \vec{x}, y) = \langle v_i, F_i(\vec{f}, \vec{x}, v_i)\rangle,$$

where $v_i \subseteq y$ maximizes $|K_{i+1}(\vec{f}, \vec{x}, v_i, F_i(\vec{f}, \vec{x}, v_i))|$, then P_i will satisfy (9) for all $v \subseteq y$. So it suffices to find BFFs $P_1, \ldots, P_{k-1}$ which satisfy (10) for all y. Define $P_1, \ldots, P_{k-1}$ as follows:

$$P_i(\vec{f}, \vec{x}, 0) = \langle 0, G_i(\vec{f}, \vec{x})\rangle,$$

$$P_i(\vec{f}, \vec{x}, y) = \begin{cases} \langle y, z_i\rangle & \text{if } |K_{i+1}(\vec{f}, \vec{x}, y, z_i)| \geq |K_{i+1}(\vec{f}, \vec{x}, \Pi_1(t_i), \Pi_2(t_i))|, \\ t_i & \text{otherwise}, \end{cases} \qquad y > 0,$$

where

$$z_i = H_i(\vec{f}, \vec{x}, y, E_1(\vec{f}, \vec{x}, \Pi_1(t_1), \ldots, \Pi_1(t_{k-1}), \Pi_2(t_1), \ldots, \Pi_2(t_{k-1})), \ldots,$$
$$E_k(\vec{f}, \vec{x}, \Pi_1(t_1), \ldots, \Pi_1(t_{k-1}), \Pi_2(t_1), \ldots, \Pi_2(t_{k-1})))$$

and

$$t_i = P_i(\vec{f}, \vec{x}, \lfloor \tfrac{y}{2} \rfloor).$$

We now show, by induction on the notation of y, that the P_i's satisfy (10) for all y. This is clear when $y = 0$. Assume that for $1 \leq i \leq k-1$, P_i satisfies (10) for $\lfloor \frac{y}{2} \rfloor$. Then for $1 \leq i \leq k-1$, P_i satisfies (9) for all $v \subseteq \lfloor \frac{y}{2} \rfloor$, so that by the preceding lemma, $z_i = F_i(\vec{f}, \vec{x}, y)$. But then, by the induction hypothesis and the definition of P_i, for $1 \leq i \leq k-1$ we have

$$P_i(\vec{f}, \vec{x}, y)$$
$$= \begin{cases} \langle y, F_i(\vec{f}, \vec{x}, y)\rangle & \text{if } |K_{i+1}(\vec{f}, \vec{x}, y, F_i(\vec{f}, \vec{x}, y))| \geq |K_{i+1}(\vec{f}, \vec{x}, v_i, F_i(\vec{f}, \vec{x}, v_i))|, \\ \langle v_i, F_i(\vec{f}, \vec{x}, v_i)\rangle & \text{otherwise}, \end{cases}$$

where $v_i \subseteq y$ maximizes $|K_{i+1}(\vec{f}, \vec{x}, v_i, F_i(\vec{f}, \vec{x}, v_i))|$. It is then clear that for $1 \leq i \leq k-1$, P_i satisfies (10) for y, as required.

It remains to show that $P_1, \ldots, P_{k-1}$ are bounded in such a way so that they are definable by WMLRN. The bound for P_1 is obtained as in the base case. Now for $2 \leq i \leq k-1$, we conclude from the definition of P_i that

$$|\Pi_2(P_i(\vec{f}, \vec{x}, y))| \leq \max_{v \subseteq y} |F_i(\vec{f}, \vec{x}, v))|.$$

We also conclude from the definition of F that

$$|F_i(\vec{f}, \vec{x}, v)| \leq |K_i(\vec{f}, \vec{x}, v, F_{i-1}(\vec{f}, \vec{x}, v))|$$

Combining these equalities with the fact that for $1 \leq i \leq k-1$, P_i satisfies (10) for all y, we see that

$$|\Pi_2(P_i(\vec{f}, \vec{x}, y))| \leq |K_i(\vec{f}, \vec{x}, \Pi_1(P_{i-1}(\vec{f}, \vec{x}, y)), \Pi_2(P_{i-1}(\vec{f}, \vec{x}, y))).$$

Finally, since $|\Pi_1(P_i(\vec{f}, \vec{x}, y))| \leq |y|$, $P_1, \ldots, P_{k-1}$ are definable by WMLRN from functionals basic feasible in $\vec{G}$, $\vec{H}$, and $\vec{K}$. □

REFERENCES

[1] A. Cobham, *The intrinsic computational difficulty of functions*, Proc. 1964 International Congress for Logic, Methodology, and the Philosophy of Science, North–Holland, Amsterdam, 1964.

[2] S. A. Cook, *Iterated recursion is* PV*-definable*, manuscript, 1989.

[3] ———, *Computability and complexity of higher-type functions*, in Logic from Computer Science, Springer-Verlag, Berlin, 1992, pp. 51–72.

[4] S. A. Cook and B. M. Kapron, *Characterizations of the basic feasible functionals of finite type*, in Feasible Mathematics, Birkhauser, Boston, 1990, pp. 71–95.

[5] H. Friedman and K. Ko, *Computational complexity of real functions*, Theoret. Comput. Sci., 20 (1982), pp. 323–352.

[6] A. Ignjatović, *On feasibility in higher types*, manuscript, 1994.

[7] B. M. Kapron, *Feasible computation in higher types*, Tech. report 249/91, Department of Computer Science, University of Toronto, Toronto, ON, 1991.

[8] B. M. Kapron and S. A. Cook, *A new characterization of Mehlhorn's poly-time functionals*, in Proc. 32nd IEEE Symposium on Foundations of Computer Science, San Juan, PR, 1991, pp. 342–347.

[9] K. Mehlhorn, *Polynomial and abstract subrecursive classes*, J. Comput. System Sci., 12 (1976), pp. 147–178.

[10] M. Townsend, *Complexity for type-*2 *relations*, Notre Dame J. Formal Logic, 31 (1990), pp. 241–262.

SIAM J. COMPUT.
Vol. 25, No. 1, pp. 133–168, February 1996

LINEAR TIME AND MEMORY-EFFICIENT COMPUTATION*

KENNETH W. REGAN†

Abstract. A realistic model of computation called the *block-move* (BM) model is developed. The BM regards computation as a sequence of finite transductions in memory, and operations are timed according to a memory cost parameter μ. Unlike previous memory-cost models, the BM provides a rich theory of linear time, and in contrast to what is known for Turing machines (TMs), the BM is proved to be highly *robust* for linear time. Under a wide range of μ parameters, many forms of the BM model, ranging from a fixed-wordsize random-access machine (RAM) down to a single finite automaton iterating itself on a single tape, are shown to simulate each other up to constant factors in running time. The BM is proved to enjoy efficient universal simulation, and to have a tight deterministic time hierarchy. Relationships among BM and TM time complexity classes are studied.

Key words. computational complexity, theory of computation, machine models, Turing machines, random-access machines, simulation, memory hierarchies, finite automata, linear time, caching

AMS subject classifications. 68Q05, 68Q10, 68Q15, 68Q68

1. Introduction.

This paper develops a new theory of linear-time computation. The *block-move* (BM) model introduced here extends ideas and formalism from the *block-transfer* (BT) model of Aggarwal, Chandra, and Snir [2]. The BT is a random-access machine (RAM) with a special *block-transfer* operation, together with a parameter $\mu : \mathbf{N} \to \mathbf{N}$ called a *memory-access cost function*. The RAM's registers are indexed 0,1,2,..., and $\mu(a)$ denotes the cost of accessing register a. A block transfer has the form

$$copy\ [a_1 \ldots b_1]\ into\ [a_2 \ldots b_2],$$

and is *valid* if these intervals have the same size m and do not overlap. With regard to a particular μ, the charge for the block transfer is $m + \mu(c)$ time units, where $c = \max\{a_1, b_1, a_2, b_2\}$. The idea is that after the initial charge of $\mu(a)$ for accessing the two blocks, a line of consecutive registers can be read or written at unit time per item. This is a reasonable reflection of how pipelining can hide memory latency, and accords with the behavior of physical memory devices (see [3], p. 1117, or [34], p. 214). An earlier paper [1] studied a model called HMM which lacked the block-transfer construct. The main memory-cost functions treated in these papers are $\mu_{\log}(a) := \lceil \log_2(a+1) \rceil$, which reflects the time required to write down the memory address a, and the functions $\mu_d(a) := \lceil a^{1/d} \rceil$ with $d = 1, 2, 3, \ldots$, which model the asymptotic increase in communication time for memory laid out on a d-dimensional grid. (The cited papers write f in place of μ and α for $1/d$.) The two-level *input/output* (*I/O*) *complexity* model of Aggarwal and Vitter [3] has fixed block size and a fixed cost for accessing the outer level, while the *uniform memory hierarchy* (UMH) model of Alpern, Carter, and Feig [5] scales block size and memory access cost upward in steps at higher levels.

The BM makes the following changes to the BT. First, the BM fixes the wordsize of the underlying machine, so that registers are essentially the same as cells on a Turing tape. Second, the BM provides native means of shuffling and reversing blocks. Third and most important, the BM allows other finite transductions S besides *copy* to be applied to the data in a block operation. A *block move* has the form

$$S\ [a_1 \ldots b_1]\ into\ [a_2 \ldots b_2].$$

*Received by the editors June 17, 1993; accepted for publication (in revised form) July 28, 1994. This research was supported in part by NSF Research Initiation Award CCR-9011248.

†Department of Computer Science, State University of New York at Buffalo, 226 Bell Hall, Buffalo, NY 14620-2000.

If x is the string formed by the symbols in cells a_1 through b_1, this means that $S(x)$ is written to the tape beginning at cell a_2 in the direction of b_2, with the proviso that a blank B appearing in the output $S(x)$ leaves the previous content of the target cell unchanged. This proviso implements *shuffle*, while *reverse* is handled by allowing $b_1 < a_1$ and/or $b_2 < a_2$. The block move is *valid* if the two intervals are disjoint and meets the *strict boundary condition* if $S(x)$ neither overflows nor underflows $[a_2 \dots b_2]$. The *work* performed in the block move is defined to be the number $|x|$ of bits read, while the *memory-access charge* is again $\mu(c)$, $c = \max\{a_1, b_1, a_2, b_2\}$. The μ*-time* is the sum of these two numbers. Adopting terms from [5], we call a BM M *memory efficient* if the total memory-access charges stay within a constant factor (depending only on M) of the work performed, and *parsimonious* if the ratio of access charges to work approaches 0 as the input length n increases.

In the BT model, Aggarwal, Chandra, and Snir [2] proved tight nonlinear lower bounds of $\Theta[n \log n]$ with $\mu = \mu_1$, $\Theta[n \log\log n]$ with $\mu = \mu_d, d > 1$, and $\Theta[n \log^* n]$ with $\mu = \mu_{\log}$, for the so-called "touch problem" of executing a sequence of operations during which every value in registers $R_1 \dots R_n$ is copied at least once to R_0. Since any access to R_a is charged the same as copying R_a to R_0, this gives lower bounds on the time for any BT computation that involves all of the input. In the BM model, however, the other finite transductions can glean information about the input in a way that *copy* cannot. Even under the highest cost function μ_1 that we consider, many interesting nonregular languages and functions are computable in linear time.

1.1. Previous models. It has long been realized that the standard unit-cost RAM model [21], [31], [18] is too powerful for many practical purposes. Feldman and Shapiro [22] contend that realistic models $\mathcal{M}$, both sequential and parallel, should have a property they call "polynomial vicinity," which we state as follows: let C be a data configuration, and let H_C stand for the finite set of memory locations (or data items) designated as "scanned" in C. For all $t > 0$, let I_t denote the set of locations (or items) i such that there exists an $\mathcal{M}$-program that, when started in configuration C, scans i within t time units. Then the model $\mathcal{M}$ has *vicinity* $v(t)$ if for all C and t, $|I_t|/|H_C| \leq v(t)$. In three-dimensional space, real machines "should have" at most cubic vicinity. The RAM model, however, has exponential vicinity even under the *log-cost criterion* advocated by Cook and Reckhow [18]. So do the random-access Turing machine (RAM-TM) forms described in [30], [26], [7], [14], [64], and so do TMs with tree-structured tapes (see [57], [63], [51], [52]). Turing machines with d-dimensional tapes (see [31], [60], [50]) have vicinity $O(t^d)$, regardless of the number of such tapes or number of heads on each tape, even with head-to-head jumps allowed. The standard TM model, with $d = 1$, has linear vicinity. The "RAM with polynomially compact memory" of Grandjean and Robson [29] limits integers i that can be stored and registers a that can be used to a polynomial in the running time T. This is not quite the same as polynomial vicinity—if $t \ll T$, the machine within t steps could still address a number of registers that is exponential in t. The BM has polynomial vicinity under μ_d (though not under $\mu_{\log}$), because any access outside the first t^d cells costs more than t time units. The theorem of [56] that deterministic linear time on the standard TM (DLIN) is properly contained in nondeterministic TM linear time (NLIN) is not known to carry over to any model of superlinear vicinity.

1.2. Practical motivations. The BM attempts to capture, with a minimum of added notation, several important properties of computations on real machines that the previous models neglect or treat too coarsely. The motivations are largely the same as those for the BT and UMH. As calibrated by μ, memory falls into a *hierarchy* ranging from relatively small amounts of low-indexed fast memory up through to large amounts of slow external storage. An algorithm that enjoys good *temporal locality of reference*, meaning that long stretches of its operation use relatively few different data items, can be implemented as a BM program that first copies the needed items to low memory (figuratively, to a cache) and is rewarded by a lower sum of memory-access charges. Good *spatial locality of reference*, meaning that needed data items are stored in neighboring locations in approximately the order of their need, is rewarded by the possibility of batching or pipelining a sequence of operations in the same block move. However, the BM appears to emphasize the sequencing of data items within a block more than the BT and UMH do, and we speak more specifically of *good serial access* rather than spatial locality of reference. The BM breaks sequential computation into phases in which access is serial and the operation is a finite transduction, and allows "random" access only between phases. Both μ*-time*(n) and the count $R(n)$ of block moves provide ways to quantify random access as a resource. The latter also serves as a measure of parallel time, since finite transductions can be computed by parallel prefix sum. Indeed, the BM is similar to the Pratt–Stockmeyer vector machine [61], and can also be regarded as a fixed-wordsize analogue of Blelloch's "scan" model [11].

1.3. Results. The first main theorem is that the BM is a very *robust* model. Many diverse forms of the machine simulate each other up to constant factors in μ-time, under a wide range of cost functions μ. Allowing multiple tapes or heads, expanding or limiting the means of tape access, allowing invalid block moves, making block boundaries preset or datadependent in a block move, and even reducing the model down to a single finite automaton that iterates itself on a single tape make no or little difference. We claim that this is the first sweeping *linear-time* robustness result for a natural model of computation. A "linear speed-up" theorem, similar to the familiar one for Turing machines, makes the constant factors on these simulations as small as desired. All of this gives the complexity measure μ*-time* a good degree of machine independence. Some of the simulations preserve the work (w) and memory-access charges (μ*-acc*) separately, while others trade w off against μ*-acc* to preserve their sum.

Section 2 defines the basic BM model and also the reduced form. Section 3 defines all the richer forms, and §4 proves their equivalence. The linear speed-up theorem and some results on memory efficiency are in §5. The second main result of this paper, in §6, shows that like the RAM but unlike what is known for the standard multitape Turing machine model (see [36], [24]), the BM carries only a constant factor overhead for universal simulation. The universal BM given is efficient under any μ_d, while separate constructions work for $\mu_{\log}$. In consequence, for any fixed $\mu = \mu_d$ or $\mu_{\log}$, the BM complexity classes DμTIME$[t]$ form a *tight deterministic time hierarchy* as the order of the time function t increases. Whether there is any hierarchy at all when μ rather than t varies is shown in §7 to tie back to older questions of determinism versus nondeterminism. This section also compares the BM to standard TM and RAM models and studies BM complexity classes. Section 8 describes open problems, and §9 presents conclusions.

2. The block-move model. We use λ for the empty string and B for the *blank* character. $\mathbf{N}$ stands for $\{0, 1, 2, 3, \dots\}$. Characters in a string x of length m are numbered $x_0x_1 \cdots x_{m-1}$. We modify the *generalized sequential machine* (GSM) of [36] so that it can exit without reading all of its input.

DEFINITION 2.1. *A* generalized sequential transducer (GST) *is a machine S with components $(Q, \Gamma, \delta, \rho, s, F)$, where $F \subseteq Q$ is the set of* terminal states, $s \in Q \setminus F$ *is the* start state, $\delta : (Q \setminus F) \times \Gamma \rightarrow Q$ *is the* transition function, *and* $\rho : (Q \setminus F) \times \Gamma \rightarrow \Gamma^*$ *is the* output function. *The* I/O alphabet Γ *may contain the blank B.*

A sequence $(q_0, x_0, q_1, x_1, \dots, q_{m-1}, x_{m-1}, q_m)$ is a halting trajectory *of S on input x if $q_0 = s$, $q_m \in F$, $x_0x_1 \dots x_{m-1}$ is an initial substring of x, and for $0 \leq i \leq m-1$, $\delta(q_i, x_i) = q_{i+1}$. The* output $S(x)$ *is defined to be $\rho(q_0, x_0) \cdot \rho(q_1, x_1) \cdots \rho(q_{m-1}, x_{m-1})$.*

By common abuse of notation we also write $S(\cdot)$ for the partial function computed by S. Except briefly in §8, all finite-state machines we consider are deterministic. A symbol c is an

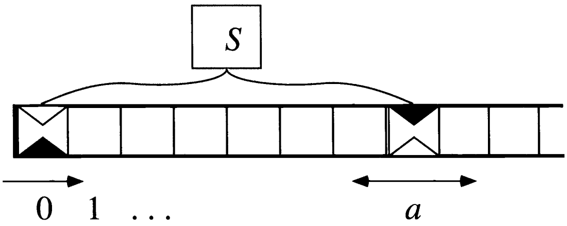


FIG. 1. *BM with allowed head motions in a pass.*

endmarker for a GST S if every transition on c sends S to a terminal state. Without loss of generality, B is an endmarker for all GSTs.

The intuitive picture of our model is a "circuit board" with GST "chips," each of which can process streams of data drawn from a single tape. The formalism is fairly close to that for Turing machines in [36].

DEFINITION 2.2. *A* block machine (BM) *is denoted by* $M = (Q, \Sigma, \Gamma, \delta, B, S_0, F)$, *where*

- Q *is a finite set consisting of* GSTs, move states, *and* halt states;
- F *is the set of halt states*;
- *every GST has one of the four labels* Ra, La, $0R$, *or* $0L$;
- move states *are labeled either* $\lfloor a/2 \rfloor$, $2a$, *or* $2a+1$;
- Σ *is the I/O alphabet of* M, *while the work alphabet* Γ *is used by all GSTs*;
- *the* start state S_0 *is a GST with label* Ra; *and*
- *the* transition function δ *is a mapping from* $(Q \setminus F) \times \Gamma$ *to* Q.

We find it useful to regard GSTs as "states" in a BM machine diagram, reading the machine in terms of the specific functions they perform, and submerging the individual states of the GSTs onto a lower level. M has two tape heads, called the "cell-0 head" and the "cell-a head," which work as follows in a GST pass (Fig. 1). Let $\sigma[i]$ stand for the symbol in tape cell i, and for $i, j \in \mathbf{N}$ with $j < i$ allowed, let $\sigma[i \dots j]$ denote the string formed by the symbols from cell i to cell j.

DEFINITION 2.3. *A* pass *by a GST S in a BM works as follows, with reference to the* current address a *and each of the four* modes Ra, La, $0R$, $0L$:

(Ra) *S reads the tape moving rightward from cell* a. *Since B is an endmarker for S, there is a cell* $b \geq a$ *in which S exits. Let* $x := \sigma[a \dots b]$ *and* $y := S(x)$. *If* $y = \lambda$, *the pass ends with no change in the tape. For* $y \neq \lambda$, *let* $c := |y| - 1$. *Then y is written into cells* $[0 \dots c]$, *except that if* $y_i = B$, *cell i is left unchanged. This completes the pass.*

(La) *S reads the tape moving leftward from cell* a. *Unless S runs off the left end of the tape (causing a "crash"), let* $b \leq a$ *be the cell in which S exits. As before let* $x := \sigma[a \dots b]$, $y := S(x)$, *and if* $y \neq \lambda$, $c := |y| - 1$. *Then formally, for* $0 \leq i \leq c$, *if* $y_i \neq B$ *then* $\sigma[i] := y_i$, *while if* $y_i = B$ *then* $\sigma[i]$ *is unchanged.*

($0R$) *S reads from cell* 0, *necessarily moving right. Let c be the cell in which S halts. Let* $x := \sigma[0 \dots c]$, $y := S(x)$, *and* $b := a + |y| - 1$. *Then y is written rightward from a into cells* $[a \dots b]$, *with the same convention about B as above.*

($0L$) *This is the same as* $0R$, *except that* $b := a - |y| + 1$, *and y is written leftward from a into* $[a \dots b]$.

Here a, b, and c are the access points *of the pass. Each of the four kinds of passes is* valid *if either* (i) $y = \lambda$, (ii) $a, b, c \leq 1$, *or* (iii) $c < \min\{a, b\}$. *The case* $y = \lambda$ *is called an* empty pass, *while if* $|x| = 1$, *then it is called a* unit pass.

In terms of §1, Ra and La execute the block move $S[a \dots b]$ *into* $[0 \dots c]$, except that the boundaries b and c are not set in advance and can depend on the data x. Similarly $0R$ and

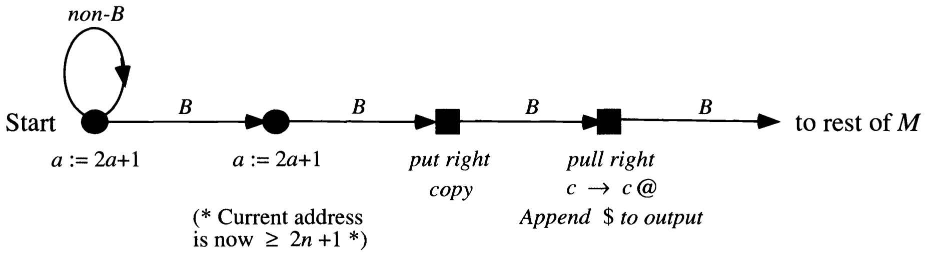


FIG. 2. *A BM that makes a fresh track.*

$0L$ execute $S\,[0\ldots c]$ *into* $[a\ldots b]$. We make the distinction that in a *pass*, the read and write boundaries may depend on the data, while in a *block move* (formalized in the next section), they are set beforehand. The tape is regarded as linear for passes or block moves but as a binary tree for addressing. The root of the tree is cell 1, while cell 0 is an extra cell above the root. The validity condition says that the intervals $[a\ldots b]$ and $[0\ldots c]$ must not overlap, with a technically convenient exception in case the whole pass is done in cells 0 and 1. If a pass is invalid, M is considered to "crash." A pass of type Ra or La figuratively "pulls" data to the left end of the tape, and we refer to it as a *pull*; similarly, we call a pass of type $0R$ or $0L$ a *put*. Furthering the analogy to internal memory or to a processor cache, these pass types might be called a *fetch* and *writeback*, respectively. An La or $0L$ pass can reverse a string on the tape.

DEFINITION 2.4. *A valid computation $\vec{c}$ by a BM $M = (Q, \Sigma, \Gamma, \delta, B, S_0, F)$ is defined as follows. Initially $a = 0$, the tape contains x in cells $0\ldots|x|-1$ with all other cells blank, and S_0 makes the first pass. When a pass by a GST S ends, let c be the character read on the transition in which S exited. Then control passes to $\delta(S, c)$. In a move state q, the new current address a' equals $\lfloor a/2\rfloor$, $2a$, or $2a+1$ according to the label of q, and letting d be the character in cell a', control passes to state $\delta(q, d)$. All passes must be valid, and a valid computation ends when control passes to a halting state. Then the* output, *denoted by $M(x)$, is defined to be $\sigma[0\ldots m-1]$, where $\sigma[m]$ is the leftmost non-Σ character on the tape. If M is regarded as an acceptor, then the language of strings accepted by M is denoted by $L(M) := \{x \in \Sigma^* | M(x)$ halts and outputs 1 $\}$.*

The convention on output is needed since a BM cannot erase, i.e., write B. Alternatively, for an acceptor, F could be partitioned into states labeled ACCEPT and REJECT.

DEFINITION 2.5. *A* memory-cost function *is any function $\mu : \mathbf{N} \rightarrow \mathbf{N}$ with the properties* (a) $\mu(0) = 0$, (b) $(\forall a)\mu(a) \le a$, *and* (c) $(\forall N \ge 1)(\forall a)\mu(Na) \le N\mu(a)$.

Our results will only require the property (c′): $(\forall N \ge 1)(\exists N' \ge 1)$ $(\forall^\infty a)$ $\mu(Na) \le N'\mu(a)$. While property (c) can be named by saying that μ is "sublinear," we do not know a standard mathematical name for (c′), and we prefer to call either (c) or (c′) the *tracking property* for the following reason.

EXAMPLE 2.1. Tracking. *Figure 2 diagrams a multichip BM routine that changes the input $x = x_0x_1\cdots x_{n-1}$ to $x_0@x_1@\cdots x_{n-2}@x_{n-1}$ @ \$, where @ acts as a "surrogate blank," and only @ or B appears to the right of the \$. This divides the tape into two* tracks *of odd and even cells. A BM can write a string y to the second track by pulling it as $By_0By_1\cdots By_{m-1}By_m$, since the blanks B leave the contents of the first track undisturbed. Two strings can also be* shuffled *this way. Since $\mu(2a) \le 2\mu(a)$, the tracking no more than doubles the memory-access charges.*

The principal memory cost functions we consider in this paper are the *log-cost function* $\mu_{\log}(a) := \lceil\log_2(a+1)\rceil$, and for all $d \ge 1$, the *d-dimensional layout function* $\mu_d(a) := \lceil a^{1/d}\rceil$. These have the tracking property.

DEFINITION 2.6. *For any memory-cost function μ, the μ-time of a valid pass that reads x and operates the cell-a head in the interval $[a \dots b]$ is given by $\mu(a) + |x| + \mu(b)$. The* work *of the pass is $|x|$, and the* memory-access charge *is $\mu(a) + \mu(b)$. A move state that changes a to a' performs 1 unit of work and has a memory-access charge of $\mu(a) + \mu(a')$. The sum of the work over all passes in a valid computation $\vec{c}$ is denoted by $w(\vec{c})$, the total memory-access charges by μ-acc$(\vec{c})$, and the total μ-time by $\mu(\vec{c}) := w(\vec{c}) + \mu\text{-}acc(\vec{c})$.*

Intuitively, the charge for a pass is $\mu(a)$ time units to access cell a, plus $|x|$ time units for reading or writing the block, plus $\mu(b)$ to communicate to the central processing unit (CPU) that the pass has ended and to reset the heads. We did not write $\max\{\mu(a), \mu(b)\}$ because b is not known until after the time to access a has already been spent; this makes no difference up to a factor of two. Replacing $|x|$ by $|x| + |S(x)|$ or by $\max\{|x|, |S(x)|\}$, or adding $\mu(c)$ to $\mu(a) + \mu(b)$, also make no difference in defining w or μ-*acc*, this time up to a constant factor that may depend on M.

DEFINITION 2.7. *For any input x on which a BM M has halting computation $\vec{c}$, we define the following complexity measures.*

Work: $w(M, x) := w(\vec{c})$.
Memory access: $\mu\text{-}acc(M, x) := \mu\text{-}acc(\vec{c})$.
μ-time: $\mu\text{-}time(M, x) := w(M, x) + \mu\text{-}acc(M, x)$.
Space: $s(M, x) :=$ *the maximum of a for all access points a in* $\vec{c}$.
Pass count: $R(M, x) :=$ *the total number of passes in* $\vec{c}$.

M is dropped when it is understood, and the above are extended in the usual manner to functions $w(n)$, μ-*acc*(n), μ-*time*(n), $s(n)$, and $R(n)$ by taking the maximum over all inputs x of length n. A measure of space closer to the standard TM space measure could be defined in the extended BM models of the next section by placing the input x on a separate read-only input tape, but we do not pursue space complexity further in this paper. The pass count appears to be sandwiched between two measures of *reversals* for multitape TMs, namely the now-standard one of [59], [35], [16] and the stricter notion of [43], which essentially counts keeping a TM head stationary as a reversal.

DEFINITION 2.8. *For any memory-cost function μ and recursive function $t : \mathbf{N} \rightarrow \mathbf{N}$,* D$\mu$TIME$[t]$ *stands for the class of languages accepted by BMs M that run in time $t(n)$, i.e., such that for all x, $\mu\text{-}time(M, x) \leq t(|x|)$.* TLIN *stands for* D$\mu_1$TIME$[O(n)]$.

We also write DμTIME$[t]$ and TLIN for the corresponding function classes. Section 7 shows that TLIN is contained in the TM linear-time class DLIN. We argue that languages and functions in TLIN have true linear-time behavior even under the most constrained implementations.

We do not separate out the work performed from the total memory-access charges in defining BM complexity classes, but do so in adapting the following notions and terms from [5] to the BM model.

DEFINITION 2.9. (a) *A BM M is* memory efficient, *under a given memory-cost function μ, if there is a constant K such that for all x, $\mu\text{-}time(M, x) \leq K \cdot w(M, x)$.*

(b) *M is* parsimonious *under μ if $\mu\text{-}time(M, x)/w(M, x) \rightarrow 1$ as $|x| \rightarrow \infty$.*

Equivalently, M is memory efficient under μ if $\mu\text{-}acc(M, x) = O(w)$ and parsimonious under μ if $\mu\text{-}acc(M, x) = o(w)$, where the asymptotics are as $|x| \rightarrow \infty$. The intuition, also expressed in [5], is that efficient or parsimonious programs make good use of a memory cache.

Definition 2.9 does not imply that the given BM M is optimal for the function f it computes. Indeed, from *Blum's speed-up theorem* [12] and the fact that μ-*time* is a complexity measure, there exist computable functions with no μ-*time*-optimal programs at all. To apply the concepts of memory efficiency and parsimony to languages and functions, we use the following relative criterion.

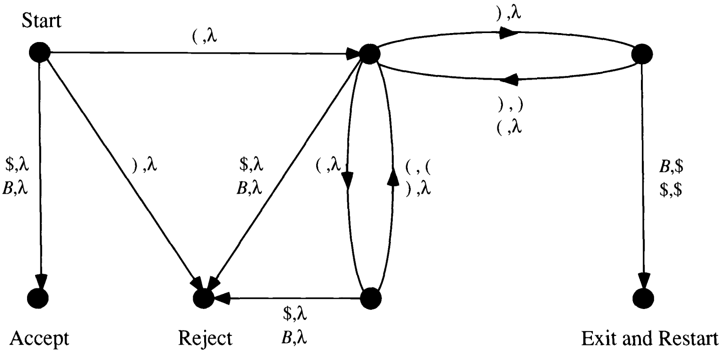


FIG. 3. *Reduced-form BM for the language of balanced parentheses.*

DEFINITION 2.10. (a) *A function f is* inherently μ-efficient *if for every BM M_0 that computes f, there is a BM M_1 that computes f and a constant $K > 0$ such that for all x, μ-time$(M_1, x) \leq K \cdot w(M_0, x)$.*

(b) *f is* inherently μ-parsimonious *if for every BM M_0 computing f there is a BM M_1 computing f such that* $\limsup_{|x| \to \infty}$ *μ-time$(M_1, x)/w(M_0, x) \leq 1$.*

By definition, μ-parsimony $\Longrightarrow$ μ-efficiency, and if f is inherently efficient (resp., parsimonious) under μ_1, then f is inherently efficient (resp., parsimonious) under every memory-cost function $\mu \leq \mu_1$.

Just for the next three examples, we drop the validity condition on rightward pulls; that is, we allow the tape intervals $[a \ldots b]$ and $[0 \ldots c]$ to overlap in an Ra move. This is intuitively reasonable so long as the cell-0 head does not overtake the cell-a head and write over a cell that the latter has not read yet. Theorem 4.1 will allow us to drop the validity condition with impunity, but the proof of Theorem 2.1 below requires that it be in force.

EXAMPLE 2.2. Balanced parentheses. *Let D_1 stand for the language of balanced parenthesis strings over $\Sigma := \{\,(,)\,\}$. Let the GST S work as follows on any $x \in \Sigma^*$: If $x = \lambda$, S goes to a terminal state marked* ACCEPT; *if x begins with "$)$", S goes to* REJECT. *Else S erases the leading "$)$" and thereafter takes bits in twos, translating*

$$(1) \qquad ((\mapsto (\qquad)) \mapsto) \qquad () \mapsto \lambda \qquad)(\mapsto \lambda.$$

If x ends in "$)$" or $|x|$ is odd, S also signals REJECT. *Then S has the property that for any $x \neq \lambda$ that it does not immediately reject, $x \in D_1 \iff S(x) \in D_1$. Furthermore, $|S(x)| < |x|/2$. We can think of D_1 as being self-reducible in a particularly sharp sense.*

Figure 3 shows the corresponding BM in the "reduced form" defined below. The "\$" endmarker is written on the first pass and prevents leftover "garbage" on the tape from interfering with later passes. We take this for granted in some later descriptions of BMs. For any memory-cost function μ, the running time of M is bounded by

$$(2) \qquad \sum_{i=0}^{\log_2 n} \mu(0) + 2^i + \mu(2^i),$$

which is $O(n)$ even for $\mu = \mu_1$. Hence the language D_1 belongs to TLIN.

EXAMPLE 2.3. Counting. *Let $\Sigma := \{\mathrm{a}, \mathrm{b}\}$. We can build a GST S with alphabet $\Gamma = \{\mathrm{a}, \mathrm{b}, 0, 1, \$, B\}$ that runs as follows on inputs of the form $x' = xu\$$ with $x \in \{\mathrm{a}, \mathrm{b}\}^*$ and $u \in \{0, 1\}^*$: S erases bits $x_0, x_2, x_4, \ldots$ of x and remembers $|x|$* modulo 2. *S then copies u, and on reading the final \$ (or on the first pass, B), S outputs $0\$$ if $|x|$ was even, $1\$$ if $|x|$*

was odd. S is also coded so that if $x = \lambda$, S goes to HALT. *Let M be the BM which iterates S on input x. Then $M(x)$ halts with $|x|$ in binary notation on its tape (followed by "\$" and "garbage"). The μ-time for this iteration is likewise $O(n)$ even for $\mu = \mu_1$.*

EXAMPLE 2.4. Simulating a TM. *Let $T := (Q, \Sigma, \Gamma, \delta, B, q_0, F)$ be a single-tape TM in the notation of* [36]. *Define the* ID alphabet *of T to be $\Gamma_I := (Q \times \Gamma) \cup \Gamma \cup \{\wedge, \$\}$, where $\wedge, \$ \notin \Gamma$. The simulating BM M on an input $x = x_0 \cdots x_{n-1}$ makes a rightward pull that lays down the delimited initial ID $\wedge(q_0, x_0)x_1x_2 \cdots x_{n-1}\$$ of $T(x)$. The finite control of T is turned into a single GST S with alphabet Γ_I that produces successive IDs in the computation with each pass. Whenever T writes a blank, M writes @. Let T be programmed to move its head to cell* 0 *before halting. Then the final pass by M removes the $\wedge$ and \$ and leaves exactly the output $y := T(x)$ on the tape. Actually, because a BM cannot erase tape cells, y would be followed by some number of symbols @, but Definition* 2.4 *still makes y the output of M. Hence the BM is a universal model of computation.*

The machines in Examples 2.2–2.4 only make rightward pulls from cell 0. Each is really a GST that iterates on its own output, a form generally known as a "cascading finite automaton" (CFA). Up to small technical differences, CFAs are comparable to the one-way "sweeping automata" studied by Ibarra et al. [37]–[41], [15]. These papers characterize both one-way and two-way arrays of identical finite-state machines in terms of these and other automata and language classes. The following shows that the BM can be regarded as a generalization of these arrays, insofar as a BM can dynamically change its origin point a and direction of operation.

DEFINITION 2.11. *The* reduced form *of the BM model consists of a single GST S whose terminal states q have labels $l_1(q) \in \{a, \lfloor a/2 \rfloor, 2a, 2a{+}1,$ HALT$\}$ and $l_2(q) \in \{Ra, La, 0R, 0L\}$. The initial pass has mode Ra with $a = 0$. Whenever a pass by S exits in some state q with $l_1(q) \neq$ HALT, the labels $l_1(q)$ and $l_2(q)$ determine the address and mode for the next pass. Computations and complexity measures are defined as before.*

THEOREM 2.1. *Every BM M is equivalent to a BM M′ in reduced form, up to constant factors in all five measures of Definition* 2.7.

Proof. The idea is to combine all the GSTs of M into a single GST S and save the current state of M in cells 0 and 1. Each pass of M is simulated by at most six passes of M', except for a "staircase" of $O(\log n)$ moves at the end which is amortized into the constant factors. This simulation expands the alphabet but does not make any new tracks. The details are somewhat delicate, owing to the lack of internal memory when a pass by M' ends, and require the validity condition on passes. The full proof is in the appendix. □

In both directions, the tape cells used by M and M' are almost exactly the same, i.e., M is simulated "in place." Hence we consider the BM and the reduced form to be essentially identical. The idea of gathering all GSTs into one works with even less technical difficulty for the extended models in the next section.

3. Extensions of the BM.

We consider five natural ways of varying the BM model: (1) Remove or circumvent the validity restriction on passes. (2) Provide "random addressing" rather than "tree access" in move states. (3) Provide delimiters a_1, b_1, a_2, b_2 for block moves $S\,[a_1 \dots b_1]$ *into* $[a_2 \dots b_2]$, where the cell b_1 in which S exits is determined or calculated in advance. (4) Require that for every such block move, b_2 is such that $S(x)$ exactly fills $[a_2 \dots b_2]$. (5) Provide multiple main tapes and GSTs that can read from and write to k-many tapes at once. These extensions can be combined. We define them in greater detail, and in the next section, prove equivalences among them and the basic model.

DEFINITION 3.1. *A* BM with buffer mechanism *has a new tape called the* buffer tape *and GST chips S with the following six labels and functions:*

(*RaB*) *The GST S reads x from the main tape beginning in cell a and writes $S(x)$ to the buffer tape. The output $S(x)$ must have no blanks in it, and it completely replaces any* *previous content of the buffer. Taking b to be the cell in which S exits, the μ-time is $\mu(a) + |x| + \mu(b)$ as before.*

(LaB) *This is defined as for RaB, but reading leftward from cell a.*

(BaR) *Here S draws its input x from the buffer, and $S(x)$ is written on the main tape starting in cell a. Blanks in $S(x)$ are allowed and treated as before. When S exits, even if it has not read all of the buffer tape, the buffer is flushed. With b the destination of the last output bit (or $b = a$ if none), the μ-time is likewise $\mu(a) + |x| + \mu(b)$.*

(BaL) *This is defined as for BaR, but writing $S(x)$ leftward from cell a.*

($0B$) *This is defined as for RaB, but using the cell-0 head to read the input, and μ-time $|x| + \mu(c)$.*

($B0$) *This is defined as for BaR, but using the cell-0 head to write the output; likewise, μ-time $|x| + \mu(c)$.*

All six types of passes are automatically valid. Further details of computations and complexity measures are the same as before. A BM with limited buffer mechanism *has no GSTs with labels $B0$ or $0B$ and consequently has no cell-0 head.*

The original BM's moves of type Ra or La can now be simulated directly by RaB or LaB followed by $B0$, while $0R$ or $0L$ is simulated by $0B$ followed by BaR or BaL. For the limited buffer mechanism, the simulation is trickier, but for $\mu = \mu_d$ we will show that it can be done efficiently. The next extension allows "random access."

DEFINITION 3.2. *The* address mechanism *adds an* address tape *and new* load *moves labeled RaA, LaA, and $0A$. These behave and are timed like the buffer moves RaB, LaB, and $0B$, respectively, but direct their output to the address tape instead. As with the buffer, the output completely replaces the previous content of the address tape. Addresses are written in binary notation with the least significant bit leftmost on the tape. The output a' of a load becomes the new current address. Move states may be discarded without loss of generality.*

EXAMPLE 3.1. Palindromes. *Let* Pal *denote the language of palindromes over a given alphabet Σ. We sketch a BM M with address mechanism that accepts* Pal. *On input x, M makes a fresh track on its tape via Example* 2.1 *and runs the procedure of Example* 2.3 *to leave $n := |x|$ in binary notation on this track. In running this procedure, we either exempt rightward pulls from the validity condition or give M the buffer mechanism as well. The fresh-track cell which divides the right half of x from the left half has address $n' := 2\lfloor n/2 \rfloor + 1$. A single $0A$ move can read n but copy the first bit as* 1 *to load the address n'. M then pokes a* \$ *into cell n'. Another load prepends a "0" so as to address cell $2n$, and M then executes a leftward pull that interleaves the left half of x with the right half. A bit-by-bit compare from cell* 0 *finishes the job. M also runs in linear μ_1-time.*

The address mechanism provides for indirect addressing via a succession of loads and makes it easy to implement pointers, linked lists, trees, and other data structures and common features of memory management on a BM, subject to charges for the number and size of the references.

Thus far, all models have allowed data-dependent block boundaries. We call any of the above kinds of BM M *self-delimiting* if there is a subalphabet Γ_e of *endmarkers* such that all GSTs in M terminate precisely on reading an endmarker. (If we weaken this property slightly to allow a GST S to exit on a nonendmarker on its second transition, then it is preserved in the proof of Theorem 2.1.) The remaining extensions preset the read block $[a_1 \dots b_1]$ and the write block $[a_2 \dots b_2]$, and this is when we speak of a *block move* rather than a *pass*. Having b_1 fixed would let us use the original GSM model from [36]. However, the machines that follow are always able to drop an endmarker into cell b_1 and force a GST S to read all of $[a_1 \dots b_1]$. Hence we may ignore the distinction and retain "GST" for consistency.

DEFINITION 3.3. *A* block move *is denoted by $S[a_1 \dots b_1]$ into $[a_2 \dots b_2]$ and has this effect on the tape: Let $x := \sigma[a_1 \dots b_1]$. Then $S(x)$ is written to the tape beginning at a_2*

and proceeding in the direction of b_2, with the proviso that each blank in $S(x)$ leaves the target cell unchanged, as in Definition 2.3. *The block move is* valid *so long as the intervals* $[a_1 \ldots b_1]$ *and* $[a_2 \ldots b_2]$ *are disjoint. It* underflows *if* $|S(x)| < |b_2 - a_2| + 1$ *and* overflows *if* $|S(x)| > |b_2 - a_2| + 1$.

By default we tolerate underflows and overflows in block moves. We draw an analogy between the next form of the BM and a text editor in which the user may mark a source and destination block and perform an operation on them. One important point is that the BM does not allow insertions and deletions of the familiar "cut-and-paste" kind; instead, the output flows over the destination block and overwrites or lets stand according to the use of B in Definition 2.3. Willard [69] describes a model of a file system that lacks insertion and deletion and gives fairly efficient algorithms for simulating them. Many text processors allow the user to define and move *markers* for points of immediate access in a file. Usually the maximum number of markers allowed is fixed to some number m. Adopting a term from data structures, we give the machine four *fingers*, with labels a_1, b_1, a_2, b_2, which can be assigned among the m markers and which delimit the source and destination blocks in any block move. Finger a_1 may be thought of as the "cursor." The dual use of "a_1" as the fixed label of a finger and as the number of the cell its assigned marker currently occupies may cause some confusion, but we try to keep the meanings clear below. The same applies to a_2, b_1, and b_2, and to later usage of these labels to name four special "address tapes."

DEFINITION 3.4. *A* finger BM *has four* fingers, *labeled* a_1, b_1, a_2, b_2, *and some number* $m \geq 4$ *of* markers. *Initially, one marker is on the last bit of the input, while all other markers and all four fingers are on the first bit in cell* 0. *An invocation of a GST S executes the block move* $S[a_1 \ldots b_1]$ *into* $[a_2 \ldots b_2]$. *The* work *performed by the block move is* $|b_1 - a_1| + 1$, *while the* memory-access charge *is* $\mu(c)$, *where* $c = \max\{a_1, b_1, a_2, b_2\}$. *In a move state,* each *marker on some cell a may be moved to cell* $\lfloor a/2 \rfloor$, $2a$, *or* $2a+1$ *(or kept where it is), and the four fingers may be redistributed arbitrarily among the markers. The cost of a move state is the maximum of* $\mu(a)$ *over all addresses a involved in finger or marker* movements; *those remaining stationary are not charged.*

One classical difference between "fingers" and "pointers" is that there is no fixed limit on the number of pointers a program can create. Rather than define a form of the BM analogous to the *pointer machines* of Schönhage and others [45], [66], [67], [49], [10], we move straight to a model that uses "random-access addressing," a mechanism usually considered stronger than pointers (for in-depth comparisons, see [9], [10] and also [68]). The following BM form is based on a random-access TM (RAM-TM; cf. "RTM" in [30] and "indexing TM" in [14], [64], [8]), and is closest to the BT.

DEFINITION 3.5. *A* RAM-BM *has one* main tape, *four* address tapes, *which are labeled* a_1, b_1, a_2, *and* b_2 *and given their own heads, and a finite control comprised of* RAM-TM states *and* GST states. *In a RAM-TM state, the current main-tape address a is given by the content of tape* a_1. *The machine may read and change both the character in cell a and those scanned on the address tapes and move each address tape head one cell left or right. In a GST state S, the address tapes give the block boundaries for the block move* $S\,[a_1 \ldots b_1]$ into $[a_2 \ldots b_2]$ *as described above, and control passes to some RAM-TM state. A RAM-TM step performs work* 1 *and incurs a memory-access charge of* $\max\{\,\mu(a), \mu(b)\,\}$, *where b is the rightmost extent of an address tape head. Block moves are timed as above. Both a RAM-TM step and a block move add* 1 *to the pass count* $R(n)$. *Other details of computations are the same as for the basic BM model.*

A fixed-wordsize analogue of the original BT model of [2] can now be had by making *copy* the only GST allowed in block moves. A RAM-BM *with address loading* can use block moves rather than RAM-TM steps to write addresses.

DEFINITION 3.6. *A finger BM or a RAM-BM obeys the* strict boundary condition *if in every block move* $S[a_1 \dots b_1]$ into $[a_2 \dots b_2]$, $|S(x)| = |b_2 - a_2| + 1$.

This constraint is notable when S is such that $|S(x)|$ varies widely for different x of the same length. The next is a catch-all for further extensions.

DEFINITION 3.7. *For* $k \geq 2$, *a* k-input GST *has* k*-many input tapes and one output tape, with* $\delta : (Q \setminus F) \times \Gamma^k \to Q$ *and* $\rho : (Q \setminus F) \times \Gamma^k \to \Gamma^*$. *Each input head advances one cell at each step.*

DEFINITION 3.8. *A* multitape BM *has some number* $k \geq 2$ *of main tapes, each possibly equipped with its own address and/or buffer tapes, and uses* k*-input GSTs in passes or block moves.*

Further details of computations and complexity measures for multitape BMs can be inferred from foregoing definitions, and various validity and boundary conditions can be formulated. The proofs in the next section will make the workings of these machines clear.

Finally, given two machines M and M' of any kind and a cost function μ, we say M' *simulates* M *linearly in* μ if μ*-time*$(M', x) = O(\mu\text{-}time(M, x)) + O(|x|)$. The extra "$O(n)$" is stated because like the RAM-TM, several BM variants give a sensible notion of computing in *sub*linear time, while all the simulations to come involve an $O(n)$-time preprocessing phase to set up tracks on the main tape. Now we can state the following theorem.

THEOREM 3.1 (main robustness theorem). *For any rational* $d \geq 1$, *all forms of the BM defined above simulate each other linearly in* μ_d*-time.*

If we adapted a standard convention for TMs to state that every BM on a given input x takes time at least $|x| + 1$ (cf. [36]), then we could say that all the simulations have constant-factor overheads in μ_d*-time*.

4. Proof of the main robustness theorem. The main problems solved in the proof are: (1) how to avoid overlaps in reading and writing by "tape-folding" (Theorem 4.1), (2) how to simulate random access with one read head whose movements are limited (Lemma 4.6), and (3) how to precompute block boundaries without losing efficiency (Lemma 4.11 through Theorem 4.15). Analogues of these problems are known in other areas of computation, but solving them with only a constant-factor overhead in μ*-time* requires some care. Some of the simulations give constant-factor overheads in both w and μ*-acc*, but others trade off the work against the memory-access charges. We also state bounds on w' and μ*-acc*$'$ for the simulating machine M' individually, and on the number R' of passes M' requires, in or after proofs. The space $s'(n)$ is always $O(s(n))$.

4.1. Simulations for data-dependent block boundaries. The first simulation uses the tracking property $\mu(Na) \leq N\mu(a)$ from Definition 2.5 and does not give constant-factor overheads in all measures. We give full details in this proof, in order to take reasonable shortcuts later.

THEOREM 4.1. *For every BM* M *with buffer, there is a BM* M' *such that for every* μ, M' *simulates* M *linearly in* μ*-time.*

Proof. Let M have the buffer mechanism. Let C be the largest number of symbols output in any transition of any GST in M. Let $K := \lceil \log_2(2C + 6) \rceil$ and $N := 2^K$. The BM M' first makes N-many tracks by iterating the procedure of Example 2.1. The track comprising cells $0, N, 2N, 3N, \dots$ represents the main tape of M, while the two tracks flanking it are "marker tracks." The track through cells $2, N + 2, \dots$ represents the buffer tape. The other tracks are an "extension track," a "holding track," C-many "pull bays," and C-many "put bays." M' uses the symbol @ to reserve free space in tracks and uses $\wedge$ and \$ to mark places in the tape. A \$ also delimits the buffer track so that leftover "garbage" does not interfere. Two invariants are that before every simulated pass by M with current address a, the current address a' of M' equals Na, and the tracks apart from the main and buffer tracks contain only blanks and @ symbols.

The move $a := 2a$ by M is simulated directly by $a' := 2a'$ in M'. The move $a := 2a+1$ is simulated by effecting $a' := \lfloor a'/2 \rfloor$ K-many times, then $a' := 2a'+1$, and then $a' := 2a'$ K-many times. The move $a := \lfloor a/2 \rfloor$ is simulated by effecting $a' := \lfloor a'/2 \rfloor$ $(K+1)$-many times, and then $a' := 2a$ K-many times. Since K is a constant, the overhead in μ-*acc* for each move is constant. Henceforth we refer to "cell a on the main track" in place of a'.

We need only describe how M' simulates each of the six kinds of pass by M. Since M has the $0B$ and $B0$ instructions, we may assume that the current address a for the other four kinds is always ≥ 1. For each state q of a GST S of M, M' has a GST S'_q which simulates S starting in state q, and which exits only on the endmarker \$. We write just S' when $q = s$ or q is understood.

(a) *RaB*. M' chooses $a_1 := 2a$, pokes $\wedge$ to the left of cell a, and pokes \$ to the left of cell a_1. M' then pulls $y_1 := S'[a \ldots a_1 - 1]$ to the C-many pull bays. By the choice of C, $|y_1| \leq Ca$, and so the pull is valid.

If the cell b in which S exits falls in the interval $[a \ldots a_1 - 1]$, then S' likewise exits in cell b. Since the exit character has no \$, the transition out of S' communicates that S has exited. M' then makes $(K+1)$-many moves $a := 2a$ so that M' now addresses cell Na_1 on the main track, which is cell $N^2 a_1$ overall. M' puts $y := y_1$ onto the extension track and then pulls y onto the buffer track. One more put then overwrites the used portion of the extension track with @ symbols. M' then effects $a := \lfloor a/2 \rfloor$ $(K+1)$-many times so that it addresses the original cell a again, and re-simulates S in order to overwrite the copy of y on the pull bays by @ symbols. All of these passes are valid. M' finally removes the $\wedge$ and \$ markers at cells a and a_1. The original time charge to M was $\mu(a) + m + (b)$, where $m = b - a + 1$. The time charged to M' in this case is bounded by

$$\begin{aligned}
\mu(Na) &+ 2 + \mu(Na-1) + \mu(Na_1) + 2 + \mu(Na_1 - 1) && \text{(poke $\wedge$ and \$)}\\
&+ \mu(Na) + Nm + \mu(Nb) && \text{(simulate S)}\\
&+ 2K\mu(N^2 a_1) && \text{(move to cell Na_1)}\\
&+ 3\mu(N^2 a_1) + 3N^2 m + 3\mu(N^2 a_1 + N^2(m-1)) && \text{(put and pull y)}\\
&+ 2K\mu(N^2 a_1) + \mu(Na) + Nm + \mu(Nb) + 2\mu(Na) + 4 + 2\mu(Na_1) && \text{(clean up)}\\
&\leq (14N + 8N^2K + 12N^2)\mu(a) + (3N^2 + 2N)m + 2N\mu(b) + 4. && (m-1 \leq a).
\end{aligned}$$

So far, both the work w' and the memory-access charges μ-*acc'* to M' are within a constant factor of the corresponding charges to M.

If S does not exit in $[a \ldots a_1 - 1]$, S' exits on the \$ marker. This tells M' to do a dummy pull to save the state q that S was in when S' hit the \$, and then to execute a put that copies y_1 from the pull bays to the put bays rightward from cell a. M' then effects $a := 2a$ so now $a = a_1$, lets $a_2 := 2a_1$, pokes another \$ to the left of cell a_2, pulls $y_2 := S'_q[a_1 \ldots a_2 - 1]$ to the pull bays, and then puts y_2 into the put bays rightward of cell a_1. Since the \$ endmarker is in cell $Na_1 - 1$, this move is valid; nor does y_2 overlap y_1. If S didn't halt in $[a_1 \ldots a_2 - 1]$, M' saves the state q' that S was in when S' hit cell a_2, setting things up for the next stage with $a_3 := 2a_2$. The process is iterated until S finally exits in some cell b in some interval $[a_{j-1} \ldots a_j - 1]$. Then $y := y_1 y_2 \cdots y_j$ equals $S[a \ldots b]$. M' moves to cell Na_j, puts y onto the extension track rightward of cell Na_j, pulls y to the buffer track, and "cleans up" the extension track as before. M' then takes $(K+1)$-many steps backward to cell a_{j-1} and cleans up the pull and put bays with a pull and a put. Finally, M' effects $a := \lfloor a/2 \rfloor$ until it finds the $\wedge$ originally placed at cell a, meanwhile removing all of the \$ markers, and then removes the $\wedge$. This completes the simulated pull by S.

Let j be such that $a_j \leq b < a_{j+1}$. Then the number m of symbols read by S is at least $a_j - a$. An induction on j shows that the running totals of both w' and μ-acc' stay bounded by Dm, where D is a constant that depends only on M, not on a or j. Hence the μ-time for the simulation by M' is within $2D$ times the μ-time charged to M for the pass. (However, when $j > 0$, μ-acc'/μ-acc may no longer be bounded by a constant.)

(b) $0B$. M' first runs S on cell 0 only and stores the output y_0 on the first cells of the C-many put bays. M' then follows the procedure for *RaB* with $a = 1$. The analysis is essentially the same.

(c) *LaB*. M' first pokes a $\wedge$ to the left of cell a and \$ to the left of cell $\lfloor a/2 \rfloor$. The $\wedge$ allows M' to detect whether a is even or odd; i.e., whether it needs to simulate $a := 2a$ or $a := 2a{+}1$ to recover cell a. M' then pulls $y_1 := S'[a \ldots \lfloor a/2 \rfloor]$ to the pull bays. Note that cell $\lfloor a/2 \rfloor$ is included; M' avoids a crash by remembering the first $2C$-many symbols of y_1 in its finite control. If S didn't exit in $[a \ldots \lfloor a/2 \rfloor]$, M' remembers the state q that S would have gone to after processing cell $\lfloor a/2 \rfloor$. M' then copies cells $[0 \ldots \lfloor a/2 \rfloor - 1]$ of the main track into cells $[\lfloor a/2 \rfloor + 1 \ldots a]$ of the holding track, and does a leftward pull by S'_q to finish the work by S, stashing its output y_2 on the put bays. If S'_q does not exit before hitting the \$, then S ran off the left end of the tape and M crashed. Let $y := y_1 y_2$. Since $|y| \leq Ca$, M' can copy y to the buffer via cell Na of the extension track by means similar to before, and "clean up" the pull and put bays and holding and extension tracks before returning control to cell a. Here both w' and μ-acc' stay within a fixed constant factor of the corresponding charges to M for the pass.

(d) *BaR*. M' marks cell a on the left with a \$, and does a dummy simulation of S on cells $[0 \ldots a - 1]$ of the buffer track. If S exits in that interval, M' puts $S[0 \ldots a - 1]$ directly onto the main track, and this completes the simulated pass. If not, M' puts $y_0 := S[0 \ldots a - 1]$ onto the holding track rightward of cell a, and remembers the state q in which S' hits the \$. M' then follows the procedure for simulating *RaB* beginning with S'_q, except that it copies $@^a y_0 y_1 \cdots y_j$ to the extension track via cell Na_j. The final pull then goes to the main track but translates @ by B so that the output written by M lines up with cell a of the main track. There is no need to "clean up" the read portion of the buffer tape since all writes to it are delimited. A calculation similar to that for *RaB* yields a constant bound on the μ-*time* and work for the simulated pass, though possibly not on the μ-*access* charges.

(e) $B0$. Under the simulation, this is the same as $0B$ with the roles of the main track and buffer track reversed and @ translated to B.

(f) *BaL*. M' marks cell a on the left with \$ and puts $y_1 := S[0 \ldots a - 1]$ *rightward* from cell a of the holding track. If S exits in that interval of the buffer tape, M' then pulls y_1 to the left end of the holding track. Note that if $|y_1| > a{+}1$, then M was about to crash. M' remembers the first symbol c' of y_1 in its finite control to keep this last pull valid just in case $|y_1| = a{+}1$. Then M' puts c' into cell a, pokes a \$ to the left of cell $\lfloor a/2 \rfloor$, and executes a "delay-1 copy" of the holding track up to the \$ into the main track leftward from cell a. If a B or @ is found on the holding track before the \$, meaning that $|y_1| \leq \lfloor a/2 \rfloor$, the copy stops there and the simulated *BaL* move is finished. If not, i.e., if $|y_1| > \lfloor a/2 \rfloor$, then the delay allows the character c'' in cell $\lfloor a/2 \rfloor - 1$ of the holding track to be suppressed when the \$ is hit, so that the copy is valid. Since $|y_1| > \lfloor a/2 \rfloor$, M' can now afford to do the following: poke a \$ to the *right* of cell a, effect $a := 2a$, and do a *leftward pull* of cells $[2a \ldots a{+}1]$ of the holding track into cells $[0 \ldots a{-}1]$ of the main track, translating @ as well as B by B to leave previous contents of the main track undisturbed. This stitches the rest of y_1 beginning with c'' correctly into place. M' also cleans up cells $[0 \ldots 2a]$ of the holding track by methods seen before, and removes the \$ signs.

If S does not exit in $[0 \ldots a - 1]$, M' executes a single Ra move starting S' from cell a, once again holding back the first character of this output y_2 just in case y_1 was empty and

$|y_2| = a+1$. If this pull is invalid, then likewise $|y_2| > a+1$ and M crashed anyway. M' then concatenates y_2 to the string y_1 kept on the holding track to form y, and does the above with y. As in *LaB*, the overhead in both w and μ-*acc* is constant. This completes the proof. □

The converse simulation of a BM by a BM with buffer is clear and has constant-factor overheads in all measures, by remarks following Definition 3.1. It is interesting to ask whether the above can be extended to a linear simulation of a *concatenable* buffer (cf. [46]), but this appears to run into problems related to the nonlinear lower bounds for the touch problem in [2]. The proof gives $w'(n) = O(w(n) + n)$ and $R'(n) = O(R(n) \log s(n))$. For μ-*acc'*, the charges in the rightward moves are bounded by a constant times $\sum_{j=0}^{\log b} \mu(b/2^j)$. For $\mu = \mu_d$ this sum is bounded by $2d\mu_d(b)$, and this gives a constant-factor overhead on μ_d-*acc*. However, for $\mu = \mu_{\log}$ there is an extra factor of $\log b$.

COROLLARY 4.2. *A BM that violates the validity conditions on passes can be simulated linearly by a BM that observes the restrictions.*

We digress briefly to show that allowing simultaneous read and overwrite on the main tape does not alter the power of the model, and that the convention on B gives no power other than *shuffle*. A *two-input Mealy machine* (2MM) is essentially the same as a 2-input GST with $\rho : (Q \setminus F) \times \Gamma^2 \to \Gamma^*$.

PROPOSITION 4.3. *Let M be a BM with the following extension to the buffer mechanism: in a put step, M may invoke any 2MM S that takes one input from the buffer and the other from the main tape, writes to the main tape, and halts when the buffer is exhausted. Then M can be simulated by a BM M' with buffer at a constant-factor overhead in all measures, for all μ.*

Proof. To simulate the put by a 2MM S, M' copies the buffer to a separate track so as to interleave characters with the segment of the main tape of M concerned. Then M' invokes a GST S' that takes input symbols in twos and simulates S. Finally M' copies the output of S' from its own buffer over the main tape segment of M. □

PROPOSITION 4.4. *At a constant-factor overhead in all measures, for all μ, a BM M can be simulated by a BM M' that lacks B but has the following implementation of shuffle: M' has the above buffer extension, but restricted to the fixed 2MM which interleaves the symbols of its two input strings.*

Proof. Let Γ' consist of Γ together with all ordered pairs of characters from Γ; then the fixed 2MM can be regarded as mapping $\Gamma^* \times \Gamma^*$ onto Γ'^*. Now consider any GST S of M that can output blanks. Let S' write a dummy character @ in place of B, and let M' shuffle the output of S' with the content of the target block of the main tape. Finally M' executes a pass which, for all $c_1, c_2 \in \Gamma$ with $c_1 \neq @$, translates $(c_1, @)$ to c_1 and (c_1, c_2) to c_2. □

Besides the tracking property, our further simulations require something which, again for want of a standard mathematical name, we call the following.

DEFINITION 4.1. *A memory-access cost function μ has the* tape-compression property *if* $(\forall \epsilon > 0)(\exists \delta > 0)(\forall^\infty a)\, \mu(\lceil \delta a \rceil) < \epsilon\, \mu(a)$.

LEMMA 4.5. *For any $d \geq 1$, the memory-cost function μ_d has the tape compression property. In consequence, $\sum_{i=0}^{\log_2 b} \mu_d(\lceil b/2^i \rceil) = O(\mu_d(b))$.*

Proof. Take $\delta < \epsilon^d$. If δ of the form $1/2^k$ satisfies (a) for $\epsilon := 1/2$, then by elementary calculation, for all but finitely many b, $\sum_{i=0}^{\log_2 b} \mu(\lceil b/2^i \rceil) \leq 2k\mu(b)$. □

Lemma 4.5 promises a constant-factor overhead on the memory-access charges for "staircases" under μ_d, whereas an extra log factor can arise under $\mu_{\log}$. The simulation of random access by tree access in the next lemma is the lone obstacle to extending the results that follow to $\mu_{\log}$. Since any function $\mu(m)$ with the tape-compression property must be $\Omega[m^\epsilon]$ for some $\epsilon > 0$, this pretty much narrows the field to the functions μ_d. To picture the tree we write UP, DOWN LEFT, and DOWN RIGHT in place of the moves $\lfloor a/2 \rfloor$, $2a$, and $2a+1$ by M'.

LEMMA 4.6. *For every BM M with address mechanism, there is a basic BM M' such that for all $d \geq 1$, M' simulates M linearly under μ_d.*

Proof. We need to show how M' simulates a load step of M that loads an address a_1 from cells $[a_0 \ldots b_0]$ of the main tape. Let $m := |a_0 - b_0| + 1$. M' makes one spare track for operations on addresses. M' first pulls a_1 in binary to the left end of this track. By Theorem 4.1 we may suppose that this pull is valid. The cost is proportional to the charge of $\mu(a_0) + m + \mu(b_0)$ to M for the load. By our convention on addresses, the least significant bit of a_1 is leftmost. In this pull, M' replaces the most significant "1" bit of a_1 by a "\$" endmarker. M' then moves UP until its cell-a head reaches cell 1. With $k := \lceil \log_2 a_0 \rceil$, the total memory-access charges so far are proportional to $\sum_{i=0}^{k} \mu(2^i)$, which is bounded by a fixed constant times $\mu(a_0)$ by Lemma 4.5. Since the number of bits in a_1 is bounded by Cm, where C depends only on M, the work done by M' is bounded by $2Cm + k$. Since $k < \mu(a_0)$, we can ignore k. Hence the μ*-time* charged so far to M' is bounded by a fixed constant of that charged to M for the load.

M' now executes a rightward pull that copies all but the bit b before the \$ endmarker, b being the second most significant bit of a_1. This pull is not valid owing to an overlap on the fresh track, but by Corollary 4.2 we may suppose that it is valid. If $b = 0$ M' moves DOWN LEFT, while if $b = 1$, M' moves DOWN RIGHT. M' then executes a put that copies the remainder of a_1 (plus the \$) rightward from the new location a. M' iterates this process until all bits of a_1 are exhausted. At the end, $a = a_1$. Because of the tracking, M' moves DOWN LEFT once more so that it scans cell $2a$, which is cell a of the main track. This completes the simulated load. Recalling $|a_1| \leq Cm$, and taking $l := \lceil \log_2(a_1) \rceil$, the μ*-time* for this second part is bounded by a constant times

$$\sum_{i=0}^{l} \mu(2^i) + 2(m - i) + \mu(2^i + 2(m - i)). \tag{3}$$

By Lemma 4.5, the total memory-access charges in this sum are bounded by a fixed constant times $\mu(a_1)$. The work to simulate the load is proportional to $m \cdot l$, that is, to $(\log a_1)^2$, which causes an extra log factor over the work by M in the load. The key point, however, is that since M loaded the address a_1, M will be charged $\mu_d(a_1)$ on the next pass, which is asymptotically greater than $(\log a_1)^2$. Hence the μ_d*-time* of M' stays proportional to the μ_d*-time* of M. □

COROLLARY 4.7. *For every BM M with both the address and buffer mechanism, we can find a basic BM M', and a BM M'' with the limited buffer mechanism, such that for any $d \geq 1$, M' and M'' simulate M linearly under μ_d.*

Proof. The constructions of Lemma 4.6 and Theorem 4.1 yield M'. For M'', we may first suppose that M is modified so that whenever M loads an address a, it first stores a spare copy of a at the left end of a special track. Now consider a pass of type $B0$ or $0B$ made by M. M'' invokes a GST that remembers cell 0 and writes 1 to the address tape. Then with $a' = 1$, M'' simulates the pass by a $Ba'R$ or $Ra'B$ move. M'' then recovers the original address a by loading it from the track. Thus far M'' is a BM with address and buffer that doesn't use its cell 0 head. The method of Lemma 4.6 then removes the address mechanism in a way unaffected by the presence of the buffer. □

We remark that Lemma 4.6 and Theorem 4.1 apply to different kinds of pass by M, with two exceptions. First, pulling 1 to the left end of the track in the proof of Lemma 4.6 may require simulating a buffer. However, this can be accounted against the cost to M for the load. Second, the buffer is needed for overlaps in the further processing of a_1. However, this is needed for at most $O(\log\log(a_1))$-many passes, each of which involves $O(\log a_1)$ work, and these costs are dominated by the time to process a_1 itself. Hence in Corollary 4.7, the bounds from Lemma 4.6 and Theorem 4.1 are additive rather than compounded, and with $\mu = \mu_d$ we obtain for M' μ_d-$acc'(n) = O(\mu_d\text{-}acc(n))$, $\mu_{\log}$-$acc'(n) = O(\mu_{\log}\text{-}acc(n) \log s(n))$, $w'(n) = O(w(n) + n + R(n) \log s(n))$, and $R'(n) = O(R(n) \log s(n))$.

LEMMA 4.8. *For every RAM-BM M, we can find a BM M' with the address and buffer mechanisms, such that for any memory-cost function μ that is $\Omega(\log n)$, M' simulates M linearly under μ.*

Proof. First, M' makes separate tracks for the address tapes and worktapes of M, and also for storing the locations of the heads on these tapes of M. Whenever M begins a block move $S[a_1 \ldots b_1]$ *into* $[a_2 \ldots b_2]$, M' first computes the signs of $b_1 - a_1$ and $b_2 - a_2$, and remembers them in its finite control. M' then loads the true address of cell a_1 on the main tape, and pulls the data through a copy of S labeled *RaB* or *LaB*—depending on sign—to the buffer. Then M' loads 0 to access a_2, loads a_2 itself, and finally copies the buffer right or left from cell a_2. Since μ is $\Omega(\log n)$, the μ-*time* charged to M' is bounded by a fixed constant times the charge of $1 + |b_1 - a_1| + \max\{\, \mu(a_1), \mu(a_2), \mu(b_1), \mu(b_2) \,\}$ incurred by M. Similarly the μ-*acc* charge to M' has the same order as that to M, though if $|b_1 - a_1| < \log(b_1)$, this may not be true of the work.

If M executes a standard RAM-TM transition, the cost to M is $1 + \mu(a_1) + \mu(c)$, where a_1 is the cell addressed on the main tape and c is the greatest extent of an address tape or worktape head of M. M' first loads a_1 and writes the symbol written by M into location a_1 with a unit put. Then M' loads each of the addresses for the other tapes of M in turn, updates each one with a unit pull and a unit put, remembers the head movement on that tape, and increments or decrements the corresponding address accordingly. The time charge for updating the other tapes stays within a fixed constant factor of $\mu(c)$. □

Remark. It would be nice to have the last simulation work when the charge to M for a RAM-TM transition is just $1 + \mu(a_1)$. The difficulty is that even though $|a_1| < a_1$, it need not hold that $c < a_1$, since M might be using a lot of space on its worktapes. The issue appears to come down to whether a multitape TM running in time t can be simulated by a BM in μ-*time* $O(t)$. We discuss related open problems in §8.

LEMMA 4.9. *A finger BM can be simulated by a BM with address and buffer mechanisms, with the same bounds as in Lemma* 4.8.

Proof. M' stores and updates the finitely many markers on separate tracks in a similar manner to the last proof. The extra work per block move simulated to write or load these addresses is $O(\log s(n))$ as before. Both here and in Lemma 4.8, $R'(n) = O(R(n))$. □

THEOREM 4.10. *Let M be a RAM-BM, a finger BM, or a BM with the address and/or buffer mechanisms. Then we can find a BM M' that simulates M linearly under any μ_d.*

Proof. This follows by concatenating the constructions of the last two lemmas with that of Corollary 4.7. Since $R'(n) = O(R(n))$ in the former, the bounds on work and pass count remain $w'(n) = O(w(n) + n + R(n) \log s(n))$ and $R'(n) = O(R(n) \log s(n))$. □

This completes the simulation of most of the richer forms of the model by the basic BM, with a constant factor overhead in μ_d-*time*. By similar means, one can reduce the number of markers in a finger BM all the way to four. In going up to the richer forms, we encounter the problem that the finger BM and RAM-BM have preset block boundaries for input, and if the strict boundary condition is enforced, also for output.

4.2. Simulations for preset block boundaries. The simulation in Theorem 4.1 does not make M' self-delimiting because it does not predetermine the cell $b \in [a_0 \ldots a_1]$ in which its own simulating GST S' will exit. We could try forcing S' to read all of $[a_0 \ldots a_1]$, but part (a) of the proof of Theorem 4.1 had $a_1 := 2a_0$, and if, e.g., $\mu(a_0) = \sqrt{a_0}$ and $b - a$ is small, M' would do much more work than it should. However, if one chooses the initial increment e to be too small in trying $a_1 := a_0 + e$, $a_2 := a_0 + 2e$, $a_3 := a_0 + 4e, \ldots$, the sum of the

μ-access charges may outstrip the work. To balance the charges we take $e := \mu(a_0)$. This requires M' to *calculate* $\mu(a)$ dynamically during its computation and involves a concept of "time-constructible function" similar to that defined for Turing machines in [36].

DEFINITION 4.2. *Let* μ *be a memory-cost function, and let* $t : \mathbf{N} \to \mathbf{N}$ *be any function. Then* t *is* μ*-time* constructible *if* $t(n)$ *is computable in binary notation by a BM* M *in* μ*-time* $O(t(n))$.

Note that the time is in terms of n, not the length of n. We use this definition for $t = \mu$ itself, in saying that μ is μ-time constructible. The following takes d to be rational because there are real numbers $d \geq 1$ such that no computable function whatever gives $\lceil m^{1/d} \rceil$ to within a factor of 2 for all m. In this section it would suffice to estimate $\lceil m^{1/d} \rceil$ by some binary number of bit length $|m|/d$, but we need the proof idea of incrementing fingers and the exact calculation of $\mu_d(m)$ for later reference.

LEMMA 4.11. *For any rational* $d \geq 1$*, the memory cost function* μ_d *is* μ_d*-time constructible by a finger BM that observes the strict boundary condition.*

Proof. For any rational $d \geq 1$, the function $\lceil m^{1/d} \rceil$ is computable in polynomial time, hence in time $(\log m)^{O(1)}$ by a single-tape TM T. The finger BM M simulates the tape of T beginning in cell 2, and tracks the head of T with its "main marker" m_1. M also uses a character @ which is combined into others like so: if T scans some character c in cell a, M scans $(c, @)$. M then uses two unit block moves $S[a \ldots a]$ *into* $[0 \ldots 0]$ and $S[0 \ldots 0]$ *into* $[a \ldots a]$ to read and write what T does. It remains to simulate the head moves by T.

To picture a tree, we again say UP, DOWN LEFT, and DOWN RIGHT in place of moves from a to $\lfloor a/2 \rfloor$, $2a$, or $2a+1$. M can test whether a is a left or right child by moving UP and DOWN LEFT and seeing whether the character scanned contains the @. If T moves right and a is a left child, M then intersperses moves UP and DOWN RIGHT with unit block moves to and from cell 0 to change $(c, @)$ back to c and place @ into cell $a+1$. If instead a is a right child, M introduces a new marker m_5 into cell 1 and writes $\wedge$ there. M moves m_5 DOWN LEFT to count how far UP m_1 has to go until it reaches either a left child or the root (i.e., cell 1). By unit block moves, M carries @ along with m_1, and by assigning a finger to marker m_5, can test whether m_5 is on cell 1. If m_1 reaches a left child, M moves it UP, DOWN RIGHT, and then DOWN LEFT until m_5 comes back to the $\wedge$. Then m_1 is in cell $a+1$. If m_1 hits the root marked by $\wedge$, then a had the form $2^k - 1$, and so M moves m_1 DOWN LEFT k times. The procedure for decrementing m_1 when T moves left is similar, with RIGHT and LEFT reversed.

For each step by T, the work by M is proportional to $\log a$. By Lemma 4.5 for μ_d, the total memory-access charge for incrementing or decrementing a finger in cell a is $O(\mu_d(a))$. Since $a \leq (\log m)^{O(1)}$, the total μ_d*-time* for the simulation is still a polynomial in $\log m$, and hence is $o(\mu_d(m))$. □

This procedure can also be carried out on one of 2^K-many tracks in a larger machine, computing $a \pm 2^K$ instead of $a \pm 1$ to follow head moves by T. The counting idea of the next lemma resembles the linear-size circuits constructed for 0–1 sorting in [55].

LEMMA 4.12. *The function* #a(x)*, which gives the number of occurrences of* "a" *in a string* $x \in \{\text{a}, \text{b}\}^*$*, is computable in linear* μ_1*-time by a BM that observes the strict boundary condition.*

Proof. The BM M operates two GSTs S_1 and S_2 that read bits of x in pairs. Each records the parity p of the number of pairs "ab" or "ba" it has seen thus far, and if $|x|$ is odd, each behaves as though the input were xb. S_1 outputs the final value of p to a second track. S_2 makes the following translations

$$\text{aa} \mapsto \text{a}, \qquad \text{bb} \mapsto \text{b}, \qquad \text{ab,ba} \mapsto \begin{cases} \text{b} & \text{if } p = 0, \\ \text{a} & \text{if } p = 1 \end{cases}$$

to form a string x' such that $|x'| = \lceil |x|/2 \rceil$. Then $\#\text{a}(x) = 2\#\text{a}(x') + p$. This is iterated until no a's are left in x, at which point the bits p combine to form #a(x) in binary notation with the least significant bit first.

M begins with one marker m_1 in cell $n-1$. We first note that even setting up the two tracks requires a trick to get two more markers to cell $n-1$. M starts a marker m_5 in cell 1 and moves it DOWN LEFT or DOWN RIGHT according to whether m_1 is on a left or right child. When m_1 reaches cell 1, m_5 records $n-1$ in reverse binary notation. Then M starts moving m_5 back up while ferrying m_2 and m_3 along with m_1. Then M places m_2 and m_3 into cells $2n$ and $4n-1$, and with reference to Example 2.1, executes $(c \mapsto c@)[0 \dots n-1]$ *into* $[2n \dots 4n-1]$ and *copy*$[2n \dots 4n-1]$ *into* $[0 \dots 2n-1]$. This also uses one increment and decrement of a marker as in the proof of Lemma 4.11.

M uses a new marker m_6 to locate where the next bit p will go, incrementing m_6 after running S_1. In running S_2, always $|S_2(x')| = \lceil |x'|/2 \rceil$, and by appropriate parity tests using its markers m_1, m_2, and m_3, M can place its fingers so that all these moves are valid and meet the strict boundary condition. For the kth iteration by S_2, these three markers are all on cells with addresses lower than $n/2^{k-2}$, and even if each needs to be incremented by 1 with the help of m_5, the μ_1 charges for simulating the iteration still total less than a fixed constant times $n/2^{k-2}$. This also subsumes the $O(\log^2 n)$ charge for updating m_6. Hence the sum over all iterations is still $O(n)$. □

THEOREM 4.13. *For every BM M and rational $d \geq 1$, we can find a finger BM M' that simulates M linearly under μ_d and observes the strict boundary condition.*

Proof. As in the proof of Theorem 4.1, let C be the maximum number of characters output in any GST transition of M, and let $K := \log_2(2C + 6)$. M' first makes $N := 2^K$ tracks, by using the last proof's modification of the procedure of Example 2.1. Besides $2C$-many tracks for handling the output of passes and one track for the main tape of M, M' uses one track to record the current address a of M with the least significant bit *rightmost*, one to compute and store $e := \mu_d(a)$ via Lemma 4.11, one to store addresses a_j below, two for Lemma 4.12, and one for other arithmetic on addresses. M' uses eight markers. Marker m_1 occupies cell Na to record the current address a of M. A move to $\lfloor a/2 \rfloor$ by M is handled by moving m_1 UP $K+1$ times and DOWN LEFT K times, and other moves are handled similarly. Meanwhile, marker m_6 stays on the last bit of the stored address a, and updating a requires only one marker increment or decrement and $O(\log \log a)$ work overall. From here on we suppress the distinction between a and Na and other details that are the same as in Theorem 4.1.

First consider a rightward pull by M that starts a GST S from cell a_0 on its main tape. M' has already stored a_0 in binary, and computes $e := \mu_d(a_0)$. Since $\mu_d(a_0) \leq a_0$, e fits to the left of marker m_6 in cell $|a|$. M' then places m_3 into cell $|a|+1$ and m_4 into cell $2|a|+1$ and executes two block moves from $[|a| \dots 0]$ and $[0 \dots |a|]$ into $[|a|+1 \dots 2|a|+1]$ that shuffle a_0 and e on their respective tracks with the least significant bits aligned and leftmost. M' then executes *add*$[|a|+1 \dots 2|a|+1]$ *into* $[|a| \dots 0]$ to produce a_1. A final carry that would make $|a_1| > |a_0|$ and cause a crash can be caught and remembered in the finite control of M' by running a "dummy addition" first and then marking cell $2|a|+1$ to suppress its output by the GST *add*. Then M' "walks" marker m_2 out to cell a_1 by using m_3 to read the value of a_1 and m_5 to increment m_3.

Next M' walks m_4 out to cell e (i.e., Ne), and keeps m_3 in cell 0. Let S' be a copy of S which pads the output of each transition out to length exactly C, and which sends its output z to the C-many tracks used as "pull bays." S' is also coded so that if S exits, S' records that fact and writes $@^C$ in each transition thereafter. Then M' can execute $S'[a_1 \dots a_2]$ *into* $[0 \dots e]$ in compliance with the strict boundary condition. Now M' can calculate the number i of non-@ symbols in z by the method of Lemma 4.12. To write the true output $y_1 = S[a_0 \dots a_1]$ and ensure the block move is valid, M' must still use the pull bays to hold y_1, so M' calculates

$i' := \lceil i/C \rceil$ (actually, $i' = N\lceil i/C \rceil$). Next M walks m_4 out to cell i' and can finally simulate the first segment of the pass by S by executing $S[a_1 \dots a_2]$ *into* $[0 \dots i']$.

If S exited in $[a_0 \dots a_1]$, M' need only transfer the output y_1 of the last pass onto the left end of the main track. This can be done in two block moves after locating markers into cells i', Ci', and $2Ci'$. Else, M' transfers y_1 instead to the put bays and assigns a new marker m_7 to the "stitch point" in the put bays for the next segment of y. The final marker m_8 goes to cell a_1 and is used for the left end of the read block in all succeeding segments. In three block moves, M' can both double e to $2e$ and compute $a_2 := a_0 + 2e$ using *add* as before. If and when the current value of e has length greater than $|a_0|$, M' reassigns marker m_6 to the end of e rather than a_0, incrementing it each time e is doubled. Then M' walks m_2 out to cell a_2 and, remembering the state q of S where the previous segment left off, produces $y_2 := S_q[a_1{+}1 \dots a_2]$ by the same counting method as before. To stitch y_2 into place on the put bays, M' converts the current location of m_7 into a numeric value k, adds it to $i := |y_2|$, and finds cells $i + k$ and $2i + k$ for two block copies. In case S did not exit in $[a_1 \dots a_2]$, m_7 is moved to cell $i + k$, m_8 to a_2, m_2 to $a_3 := a_0 + 4e$, and the process is repeated.

Let b be the actual cell in which S exits, and let $j \geq 0$ be such that $a_j < b \leq a_{j+1}$. Then the μ_d*-time* charged to M for the pull is at least

$$t_j := \mu_d(a_0) + \mu_d(a_0 + 2^{j-1}e) + 2^{j-1}e \geq 2e + 2^{j-1}e. \tag{4}$$

(For $j = 0$, read "2^{j-1}" as 0.) By Lemma 4.5, the memory access charge for walking a marker out to cell a_j is bounded by a constant (depending only on d) times $\mu_d(a_j)$. The charges for the marker arithmetic come to a polynomial in $\log a_j$, and the charges for stitching segments y_j into place stay bounded by the work performed by M'. Hence the μ_d*-time* charged to M' is bounded by a constant times

$$u_j := \mu_d(a_0) + \sum_{i=0}^{j} \mu_d(a_0 + 2^i e) + e + \sum_{i=0}^{j-1} 2^i e. \tag{5}$$

Then $u_j \leq e + \sum_{i=0}^{j} \mu_d(2^{i+1}a_0) + 2^j e \leq 2^{j+2}e + 2^j e \leq 10t_j$.

For a leftward pull step by M, M' uses the same choice of $e := \mu_d(a_0)$. If $e > a_0/2$, then M' just splits $[0 \dots a_0]$ into halves as in the (*LaB*) part of the proof of Theorem 4.1. Else, M' proceeds as before with $a_{j+1} := a_0 - 2^j e$ and checks at each stage whether $a_{j+1} \geq a_0/2$ so that the next simulated pull will be valid. If not, then the amount of work done by M thus far, namely $2^{j-1}e$, is at least $a_0/4$. Thus M' can copy all of $[a_j \dots 0]$ to another part of the tape and finish it off while remaining within a constant factor of the charge to M. The remaining bounds are much the same as those for a rightward pull above.

For a rightward or leftward put, marker m_1 is kept at the current address a, cell 0 is remembered in the finite control, and the procedure for a rightward pull is begun with $a_0 = 1$ and m_8 assigned there. Here $e = 1$, and the rest is a combination of the (*BaR*) or (*BaL*) parts of the proof of Theorem 4.1 to ensure validity, and the above ways to meet the strict boundary condition in all block moves. □

Remarks. This simulation can be made uniform by providing d as a separate input. It can also be done using 8 tracks rather than $2C + 6$, though even taking $e := \mu_d(a_0)/C$ does not guarantee that the *third* stage of a rightward pull, which reads $[a_0 + 2e, a_0 + 4e]$, will be valid. The fix is first to write the strings y_j further rightward on the tape, then assemble them at the left end. Theorem 4.13 preserves $w(n) + \mu_d\text{-}acc(n)$ up to constant factors, but does not do so for either $w(n)$ or $\mu_d\text{-}acc(n)$ separately. When $d < 1$, the case $b = a$ gives a worst-case extra work of $a^{1/d}$, while the case of $b = 2a$ gives a total memory-access charge of roughly $2(\log a)(d - 1)/d$ times $\mu_d(a)$. This translates into $w'(n) = O(w(n) + n + R(n)s(n)^{1/d})$

and μ_d-$acc'(n) = O(\mu_d\text{-}acc(n) \log s(n))$. However, when $d = 1$, *both* w and μ_1-*acc* are preserved up to a factor of $10N$. Allowing that $\mu_d(a_0)$ can be estimated to within a constant factor in $O(\log a_0)$ block moves, the pass count still carries $R'(n) = O(R(n)\log^2 s(n))$ because each movement in walking a marker to a_j adds 1 to R'. The following shows some technical improvements of having addressing instead of tree access.

THEOREM 4.14. *Let $\mu = \mu_{\log}$ or $\mu = \mu_d$ with d rational. Then every BM M can be simulated linearly under μ by a RAM-BM M' with address loading that observes the strict boundary condition.*

Proof. For μ_d the simulation of the finger BM M' from the last proof by a RAM-BM is clear—the RAM-BM can even use RAM-TM steps for the address arithmetic. For $\mu_{\log}$, the point is that M' can take $e := |a_0|$, and we may presume e is already stored. The calculated quantities a_j can be loaded in one block move. (Using RAM-TM steps to write them would incur $\mu_{\log}$ access charges proportional to $\log a_0 \log\log a_0$.) The tradeoff argument of the proof of Theorem 4.13 works even for $\mu_{\log}$, and the above takes care of a constant-factor bound on the other steps in the simulation. This also gives $R'(n) = O(R(n)\log s(n))$. □

The tradeoff method of Theorem 4.13 seems also to be needed for the following "tape-reduction theorem."

THEOREM 4.15. *For every rational $d \geq 1$, a multitape BM M can be simulated linearly in μ_d-time by a one-tape BM M'.*

Proof. Suppose that M uses k tapes, each with its own buffer, and GSTs S that produce k output strings as well as read k inputs. We first modify M to a machine M' that has k main tracks, k address tracks, one "input track," and one "buffer track." For any pass by M with S, M' will interleave the k inputs on the input track, do one separate pull for each of the k outputs of S, and interleave the outputs on its buffer track. When M subsequently invokes a k-input GST T to empty its buffers, M' uses a one-tape GST that simulates T on the buffer track, invoking it k times to write each of the k outputs of T to their destinations on the main tracks.

It remains only to show how M' marks the portions of the inputs to interleave. As in the proof of Theorem 4.13, there is the difficulty of not knowing in advance how long S will run on its k inputs. The solution is the same. M' first calculates the maximum a_j of the addresses $a_1, \ldots, a_k$ on its address tracks and then calculates $e := \mu_d(a_j)$. For each i, $1 \leq i \leq k$, M' drops an endmarker into cell $a_i \pm e$ according to the direction on main track i. Then M' copies only the marked-off portions of the tracks, putting those on its input track, and simulates the one-tape version S_1 of S. If S_1 exits within that portion, then M' continues as M' does. If S_1 does not exit within that portion, M' tries again with $a_i \pm 2e$, $a_i \pm 4e, \ldots$ until it does. The same calculation as in Theorem 4.13, plus the observation that if the direction on track j is leftward then no track uses an address greater than $2a_j$, completes the proof. □

Finally, we may restate Theorem 3.1 in a somewhat stronger form.

THEOREM 4.16. *For any rational $d \geq 1$, all of the models defined in* §3 *are equivalent, linearly in μ_d-time, to a BM in reduced form that is self-delimiting with* "$" *as its only endmarker.*

Proof. This is accomplished by Theorems 2.1 through 4.15. The procedures of Lemmas 4.13 and 4.6 and Theorem 4.1 are self-delimiting and need only one endmarker $. The trick of writing $ on special tracks into the cell immediately left or right of the addressed cell a allows $ to survive the proof of Theorem 2.1 without being "tupled" into the characters c_0, c_1, or c_a. □

With all this said and verified, we feel justified in claiming that there is one salient Block Machine model, and that the formulations given here are natural. The basic BM is the tightest for investigating the structure of computations, and helps the lower bound technique we suggest in Section 8. The richer forms make it easier to show that certain functions do belong to Dμ_dTIME[$t(n)$].

5. Linear speed-up and efficiency. The following "linear speed-up" theorem shrinks the constants in all the above simulations, at the usual penalty in alphabet size. First, we give a precise definition.

DEFINITION 5.1. *The* linear speed-up property *for a model of computation and measure of time complexity states that for every machine M with running time $t(n)$, and every $\epsilon > 0$, there is a machine M' that simulates M and runs in time $\epsilon \cdot t(n) + O(n)$.*

In the corresponding definition for TMs in [36], the additive $O(n)$ term is $n+1$ and is used to read the input. For the DTM, time $O(n)$ properly contains time $n+1$, while for the NTM these are equal [13]. For the BM under cost function μ, the $O(n)$ term is $n + \mu(n)$.

THEOREM 5.1. *With respect to any unbounded memory cost function μ that has the tape compression property, all of the BM variants described in* §§2 *and* 3 *have the linear speed-up property.*

Proof. Let the BM M and $\epsilon > 0$ be given. The BM M' uses two tracks to simulate the main tape of M. Let δ in the tape-compression property be such that for almost all n, $\mu(\delta n) \leq (\epsilon/12C) \cdot \mu(n)$. Here C is a constant that depends only on M. Let $k := \lceil 1/2\delta \rceil$, let "@" stand for the blank in Γ, and let $\Gamma' := \Gamma^k \cup \{ B \}$. M' uses B only to handle its own two tracks. We describe M' as though it has a buffer; the constant C absorbs the overhead for simulating one if M' lacks the buffer mechanism. On any input x of length n, M' first spends $O(n)$ time units on a pull step that writes x into $\lceil n/k \rceil$-many characters over the compressed alphabet Γ' on the main track. Thereafter, M' simulates M with compressed tapes. In any pass by M that writes output to the main tape, M' writes the compressed output to the alternate track. M' then uses the pattern of @ symbols in each compressed output character to mask the elements of each main track character that should not be overwritten, sending the combined output to the buffer. One more pass writes the result back to the main tape. If the cost to M for the pass was $\mu(a) + |b-a| + \mu(b)$, the cost to M', allowing for the tracking, is no more than

$$\begin{aligned} &3\,[\,\mu(2\lceil a/k \rceil) + (2/d)|b-a| + 2 + \mu(2\lceil b/k \rceil)] \\ &\quad \leq (\epsilon/2)\mu(a) + (\epsilon/2)|b-a| + 6 + (\epsilon/2)\mu(b). \end{aligned}$$

The "+2" and "+6" allow for an extra cell at either end of the compressed block. Since μ is unbounded, we have $\mu(a) \cdot (\epsilon/2) + 6 \leq \epsilon \cdot \mu(a)$ for all but finitely many a. The main technical difficulty of the standard proof for TMs is averted because μ absorbs any time that M might spend moving back and forth across block boundaries. The compression by a factor of ϵ holds everywhere except for cells $1, \ldots, m$ on the main tape, where m is least such that $\mu(m) \geq 12/\epsilon$, but M' can keep the content of these cells in its finite control. The remaining details are left to the reader. For BMs with address tapes, we may suppose that the addresses are written in a machine-dependent radix rather than in binary. □

COROLLARY 5.2. *For all of the simulations in Theorems* 2.1–4.15, *and all* $\epsilon > 0$,

(a) *if M runs in μ_d-time $t(n) = \omega(n)$, then M' can be constructed to run in μ_d-time $\epsilon t(n)$ for all but finitely many n.*

(b) *if M runs in μ_d-time $O(n)$, then M' can be made to run in μ_d-time $(1+\epsilon)n$.*

Mostly because of Lemma 4.6 and Theorem 4.13, the above simulations do not guarantee constant factor overheads in either w or μ-*acc*. They do, however, preserve μ-efficiency.

PROPOSITION 5.3. *For all of the simulations of a machine M by a machine M' in Theorems* 2.1–4.15, *and memory cost functions μ they hold for, if M is μ-efficient then M' is also μ-efficient.*

Proof. Let K_1 be the constant from the simulation of M by M', and let K_2 come from Definition 2.9(a) for M. Then for all but finitely many inputs x, we have

$$\mu\text{-}time(M', x) \leq K_1(\mu\text{-}time(M, x) + |x|) \leq K_1(K_2(w(M, x) + |x|) \leq 2K_1K_2w(M', x).$$

The last inequality follows because every simulation has $w(M', x) \geq w(M, x)$ and $w(M', x) \geq |x|$. Hence M' is μ-efficient. □

So long as we adopt the convention that every function takes work at least n+1 to compute, we can state the following corollary.

COROLLARY 5.4. *For any memory-cost function μ_d, with $d \geq 1$ and rational, the notion of a language or function being memory efficient under μ_d does not depend on the choice among the above variants of the BM model.* □

We do not have analogous results for parsimony. However, the above allows us to conclude that for $d = 1, 2, 3, \ldots$, memory efficiency under μ_d is a fundamental property of languages or functions. Likewise we have a robust notion of the class $\mathrm{D}\mu_d\mathrm{TIME}[t(n)]$ of functions computable in *μ_d-time* $t(n)$, for any time bound $t(n) \geq n$. The next section shows that for any fixed d, the classes $\mathrm{D}\mu_d\mathrm{TIME}[t(n)]$ form a tight hierarchy as the time function t varies.

6. Word problems and universal simulation. We use a simple representation of a list $\vec{x} := (x_1, \ldots, x_m)$ of nonempty strings in Σ^* by the string $x_1\#\ldots\#x_m\#$, where $\# \notin \Sigma$. More precisely, we make the last symbol c of each element a pair $(c, \#)$ so as to separate elements without adding space, and also use pair characters $(c, @)$ or $(c, \$)$ to mark selected elements. The *size* of the list is m, while the *bit length* of the list is $n := \sum_{i=1}^{m} |x_i|$. We let r stand for $\max\{\, |x_i| : 1 \leq i \leq m \,\}$. Following [16] we call the list *normal* if the strings x_i all have length r. We number lists beginning with x_1 to emphasize that the x_i are not characters.

LEMMA 6.1. (a) *The function mark($\vec{x}$, y), which marks all occurrences of the string y in the normal list $\vec{x}$, belongs to* TLIN.

(b) *The function shuffle, which is defined for normal lists $\vec{x} := (x_1, \ldots, x_m)$ and $\vec{y} := (y_1, \ldots, y_m)$ of the same length and element size r by shuffle($\vec{x}$, $\vec{y}$) $= (x_1, y_1, x_2, y_2, \ldots, x_m, y_m)$, belongs to* TLIN. *Here r as well as m may vary.*

Remark. Even if the lists $\vec{x}$ and $\vec{y}$ are not normal, *mark* and *shuffle* can be computed in linear *μ_1-time* so long as they are *balanced* in the sense that $(\exists k)(\forall i)2^{k-1} < |x_i| \leq 2^k$. This is because a balanced list can be padded out to a normal list in linear *μ_1-time* (we do not give the details here), and then the padding can be removed. To normalize an unbalanced list may expand the bit length quadratically, and we do not know how to compute *shuffle* in linear *μ_1-time* for general lists.

Proof. (a) Let r be the element size of the normal list $\vec{x}$. If $|y| \neq r$, then there is nothing to do. Else, the BM M uses the idea of "recursive doubling" (cf. the section on vector machines in [6]) to produce y^k, where $k = \lceil \log_2 m \rceil$. This time is linear as a function of $n = rm$. Then M interleaves $\vec{x}$ and y^k on a separate track, and a single pass that checks for matches between # signs marks all the occurrences of y in $\vec{x}$ (if any).

(b) Suppose m is even. M first uses two passes to divide $\vec{x}$ into the "odd list" $x_1@^r x_3 @^r \cdots x_{m-1}@^r$ and the "even list" $@^r x_2 @^r x_4 @^r \cdots @^r x_m$. Single passes then convert these to $x_1 @^{3r} x_3 @^{3r} \cdots x_{m-1} @^{3r}$ and $@^{2r} x_2 @^{3r} x_4 @^{3r} \cdots @^{3r} x_m$. A pull step that writes the second over the first but translates @ to B then produces $\vec{x}\,' := x_1 @^r x_2 @^r x_3 @^r \cdots @^r x_m$. If m is odd then the "odd list" is $x_1 @^r x_3 @^r \cdots @^r x_m$ and the "even list" is $@^r x_2 @^r x_4 @^r \cdots @^r x_{m-1} @^r$, but the final result $\vec{x}\,'$ is the same. By a similar process, M converts $\vec{y}$ to $\vec{y}\,' := @^r y_1 @^r y_2 \cdots @^r y_m @^r$. Writing $\vec{y}\,'$ on top of $\vec{x}\,'$ and translating @ to B then yields *shuffle*($\vec{x}$, $\vec{y}$). This requires only a constant number of passes. □

A *monoid* is a set H together with a binary operation $\circ$ defined on H, such that $\circ$ is associative and H has an element that is both a right and a left identity for $\circ$. We fix attention on the following representation of the *monoid of transformations* $\mathcal{M}_S$ of a finite-state machine S. $\mathcal{M}_S$ *acts* on the state set Q of S and is *generated* by the functions $\{ g_c : c \in \Sigma \}$, defined by $g_c(q) = \delta(q, c)$ for all $q \in Q$, by letting $\circ$ be composition of maps on Q, and closing out the g_c under $\circ$. Here we ignore the output function ρ of S, intending to use it once the *trajectory* of states S enters on an argument z is computed. We also remark that $\mathcal{M}_S$ need not contain the identity mapping on Q, though it does no harm for us to adjoin it. By using known decomposition theorems for finite transducers [47], [32], [48], we could restrict attention to the cases where each g_c either is the identity on Q or identifies two states (a "reset machine") or each g_c is a permutation of Q and $\mathcal{M}_S$ is a group (a "permutation machine"; cf. [17]). These points do not matter here. We encode each state in Q as a binary string of some fixed length k, and encode each element g of $\mathcal{M}_S$ by the list $q\#g(q)\#\cdots$ over all $q \in Q$. Without loss of generality, we extend Q to $Q' := \{ 0, \ldots, 2^k-1 \}$ and make g the identity on elements $q \geq n$.

The *word problem* for monoids is as follows: given a list $\vec{g} := g_n g_{n-1} \cdots g_2 g_1$ of elements of the monoid, not necessarily distinct, compute the representation of $g_n \circ g_{n-1} \circ \cdots \circ g_2 \circ g_1$. Let us call the following the *trajectory problem*: given $\vec{g}$ and some $w \in \{ 0, 1 \}^k$, compute the n-tuple $(g_1(w), g_2(g_1(w)), \ldots, \vec{g}\,(w))$. The basic idea of the following is that "parallel prefix sum" is μ_1-efficient on a BM.

LEMMA 6.2. *There is a fixed BM M that, for any size parameter k, solves the word and trajectory problems for monoids acting on* $\{ 0, 1 \}^k$ *in* μ_1*-time* $O(n \cdot k2^k)$. *In particular, these problems for any fixed finite monoid belong to* TLIN.

Proof. Let T be a TM which, for any k, composes two mappings $h_1, h_2 : \{ 0, 1 \}^k \to \{ 0, 1 \}^k$ using the above representation. For ease of visualization, we make T a single-tape TM which on any input of the form $h_2\#h_1\#$ uses only the $2k \cdot 2^k$ cells occupied by the input as workspace, and which outputs $h_2 \circ h_1\#$ shuffled with "@" symbols so that the output has the same length as the input. We also program T so that on input $h\#$, T leaves h unchanged. The running time $t(k)$ of T depends only on k and is $O(k2^k)^2$. As in Example 2.4, we can create a GST S whose input alphabet is the ID alphabet of T, such that for any nonhalting ID I of T, $S(I)$ is the unique ID J such that $I \vdash_T J$.

The BM M operates as follows on input $\vec{g} := g_n\#g_{n-1}\#\cdots\#g_2\#g_1\#$. It first saves $\vec{g}$ in cells $[(nk \cdot 2^k + 1) \ldots (2nk \cdot 2^k)]$ of a separate storage track. We may suppose that n is even; if n is odd, g_n is left untouched by the current phase of the recursion. M first sets up the initial ID of T on successive pairs of maps, viz. $\wedge q_0 g_n\#g_{n-1}\# \wedge q_0 g_{n-2}\#g_{n-3}\# \cdots \wedge q_0 g_2\#g_1\#$. Then M invokes S in repeated left-to-right pulls, until all simulated computations by T have halted. Then M erases all the @'s, leaving $(g_n \circ g_{n-1})\#(g_{n-2} \circ g_{n-3})\# \cdots (g_2 \circ g_1)\#$ on the tape. The number of sweeps is just $t(k)$, and hence the total μ_1*-time* of this phase is $\leq 2t(k) \cdot n = O(n)$.

M copies this output to cells $[((n/2)k \cdot 2^k + 1) \ldots (nk \cdot 2^k)]$ of the storage track, and then repeats the process, until the last phase leaves $h := g_n \circ g_{n-1} \circ \cdots \circ g_2 \circ g_1$ on the tape. Since the length of the input halves after each phase, the total μ_1*-time* is still $O(n)$. This finishes the word problem.

To solve the trajectory problem, M uses the stored intermediate results to recover the path $(w, g_1(w), g_2(g_1(w)), \ldots, h(w)) =: (w, w_1, w_2, \ldots, w_n)$ of the given $w \in \{ 0, 1 \}^k$. Arguing inductively from the base case $(w, h(w))$, we may suppose that M has just finished computing the path $(w, w_2, w_4, \ldots, w_{n-2}, w_n)$. M shuffles this with the string $g_1\#g_3\#g_5\#\ldots\#g_{n-1}$ and then simulates in the above manner a TM T' that given a g and a w computes $g(w)$. All this takes μ_1*-time* $O(n)$. □

The following presupposes that all BMs M are described in such a way that the alphabet Γ_M of M can be represented by a uniform *code* over $\{ 0, 1 \}^*$. This *code* is extended to represent monoids $\mathcal{M}$ as described above.

THEOREM 6.3. *There is a BM M_U and a computable function code such that for any BM M and rational $d \geq 1$, there is a constant K such that for all inputs x to M, M_U on input $(code(M), code(x), d)$ simulates $M(x)$ within μ_d-time $K \cdot \mu_d$-time(M, x).*

Proof. M_U uses the alphabet $\Gamma_U := \{0, 1, @, \$, (0, \#), (1, \#), (@, \#), \Delta, B\}$. By Theorem 2.1, we may suppose that M has a single GST $S = (Q, \Gamma_M, \Gamma_M, \delta, \rho, s_0)$. Let $k := \lceil \log_2 |\Gamma_M| \rceil$, and let l be the least integer above $\log_2 |Q|$ that is a multiple of k. The *code* function on strings codes each $c \in \Gamma_M$ by a 0–1 string of length k, except that the last bit of $code(c)$ is combined with # and B is coded by $@^{k-1}(@, \#)$.

The monoid $\mathcal{M}$ of transformations of S is encoded by a k-tuple of elements of the form $code(c)code(g_c)$ over all $c \in \Gamma_M$. Here $code(g_c)$ is as described before Lemma 6.2. Dummy states are added to Q so that $code(g_c)$ has length exactly $2l \cdot 2^l$; then $code(\mathcal{M})$ has length exactly $2^k(k + 2l \cdot 2^l)$. Let C be the maximum number of symbols written in any transition of M. The code of S includes a string $code(\rho)$ that gives the output for each transition in δ, padded out with @ symbols to length exactly C (i.e., length Ck under *code*). The rest of the code of M lists the mode-change information for each terminal state of S. Finally, the input x to M is represented by the string $code(x)$ of length $|x|2^k$.

M_U has four tracks: one for the main tape of M, one for the code of M, one for simulating passes by M, and one for scratchwork. M_U uses d to compute $e := \mu_d(a)$, and follows just the part of the proof of Theorem 4.13 that locates the cells $a_j := a \pm 2^{j-1}e$, in order to drop \$ characters there. This allows M_U to pull off from its main track in cells $[a \ldots a_j]$ the *code* of the first $m := 2^{j-1}e/4k$ characters of the string x that M reads in the pass being simulated. (If this pass is a put rather than a pull, then $e = 1$ and x is in cells $[1 \ldots 2^{j-1}]$.) Then M_U changes $code(z)$ to

$$z' := (code(z_0))^j \cdot (code(z_1))^j \cdots (code(z_{m-1}))^j,$$

where $m := |z|$ and $j := 2^k(1 + 2(l/k)2^l)$. This can be done in linear μ_1-*time* by iterating the procedure for *shuffle* in Lemma 6.1(b). Now for each $i, 0 \leq i \leq m-1$, the ith segment of z' has the same length as $code(\mathcal{M})$. Next, M uses "recursive doubling" to change $code(\mathcal{M})$ to $(code(\mathcal{M}))^m$. This also takes only $O(m)$ time. Then the strings z' and $(code(\mathcal{M}))^m$ are interlaced on the scratchwork track. A single pass that matches the labels $code(c)$ to segments of z' then pulls out the word $g_z := g_{z_0} \cdot g_{z_1} \cdots g_{z_{m-1}}$.

M evaluates this word by the procedure of Lemma 6.2, yielding the encoded trajectory $s' := (s_0, s_1, \ldots, s_m)$ of S on input z. By a process similar to that of the last paragraph, M_U then aligns s' with $(code(\rho))^m$ and interleaves them, so that a single pass pulls out the output y of the trajectory. Then $code(y)$ is written to the main tape, erasing the symbols Δ used for padding and translating @ to B. The terminal state s_m of the trajectory is matched against the list that gives the mode information for the next pass of M (Lemma 6.1(a)), and M_U changes its mode and/or current address accordingly.

If the original pass by M cost μ-*time* $\mu(a) + m + \mu(b)$, then the simulation takes μ-*time* $\mu(4a) + O(m) + \mu(4b)$. The constant in the "$O(m)$" depends only on M. We have described M_U as though there were no validity restrictions on passes, but Theorems 4.1 and 2.1 convert M_U to a basic BM while keeping the constant overhead on μ_d-*time*. □

Remarks. This result implies that there is a fixed, finite collection of GSTs that form an efficient "universal chipset." It might be interesting to explore this set in greater detail. The constant on the "$O(m)$" is on the order of $2^{2(l+k)}(l + k)$. We inquire whether there are other representations of finite automata or their monoids that yield notably more efficient off-line simulations than the standard one used here. The universal simulation in Theorem 6.3 does not preserve w or μ_d-*acc* individually because it uses the method of Theorem 4.13 to compensate for its lack of "foreknowledge" about where a given block move by M will exit. The simulation does preserve memory efficiency, on account of Proposition 5.3. If, however, we suppose that M is already self-delimiting in a way made transparent by *code*, then we obtain constant overheads in both w and μ-*acc*, and the simulation itself becomes independent of μ.

THEOREM 6.4. *There is a BM M_U and a computable function code such that for any memory-cost function μ and any self-delimiting BM M, there is a constant K such that for all inputs x to M, M_U on input $x' = (code(M), code(x))$ simulates $M(x)$ with $w(U, x') \leq Kw(M, x)$ and $\mu\text{-}acc(U, x') \leq K\mu\text{-}acc(M, x)$.*

Proof. The function *code* is changed so that it encodes the endmarkers of M by strings that begin with "\$." Then M_U pulls off the portion x of its main track up to \$. The rest of the operation of M_U is the same, and the bounds now require only the tracking property of μ. (If the notion of "self-delimiting" is weakened as discussed before Definition 3.3, then we can have M_U first test whether a GST S exits on the second symbol of x.) □

To use these results for diagonalization, we need two preliminary lemmas. Recall that a function t is μ-*time constructible* if $t(n)$ is computable in binary notation in μ-time $O(t(n))$. Since all of n must be read, t must be $\Omega(\log n)$.

LEMMA 6.5. *If a BM M is started on an input of length n, then any pass by M either takes μ_1-time $O(n)$ or else no more than doubles the accumulated μ-time before the pass.*

Proof. Any portion of the tape other than the input that is read in the pass must have been previously written in some other pass or passes. (Technically, this uses our stipulation that B is an endmarker for GSTs.) Thus the conclusion follows. □

LEMMA 6.6. *For any memory-cost function μ that is μ-time constructible, a BM M can maintain a running total of its own μ-time with only a constant-factor slowdown.*

Proof. To count the number $m = |b - a| + 1$ of transitions made by one of its GST chips S in a given pass, a BM M can invoke a "dummy copy" of S that copies the content x of the cells up to where S exits to a fresh track, and then count $|x|$ on that track by the $O(m)$-time procedure of Example 2.3. Then M invokes S itself and continues operation as normal. Since μ is $\Omega(\log n)$, the current address a can be copied and updated on a separate track in μ-*time* $O(\mu(a))$. Also in a single pass, M can add a and m in μ-*time* $O(\mu(a) + m)$, and thus obtain b itself. M then calculates $\mu(b)$ in μ-*time* $O(\mu(b))$, and finally adds $k := \mu(a) + m + \mu(b)$ to its running total t of μ-*time*. In case t is much longer than k, we want the work to be proportional to $|k|$, not to $|t|$. Standard "carry–save" techniques, or alternatively an argument that long carries cannot occur too often, suffice for this. □

THEOREM 6.7. *Let $d \geq 1$ be rational, and let t_1 and t_2 be functions such that t_2 is μ_d-time constructible, and t_1 is $o(t_2)$. Then* DμTIME$[t_1]$ *is properly contained in* DμTIME$[t_2]$.

Proof. The proof of Theorem 6.3 encoded BMs M over the alphabet Γ_U, but let *code*$'$ recode M over $(00 \cup 11)^*$. We build a BM M_D that accepts a language $D \in$ DμTIME$[t_2] \setminus$ DμTIME$[t_1]$ as follows. M_D has two extra tracks on which it logs its own μ-*time*, as in Lemma 6.6. On any input x, M_D first calculates $n := |x|$, and then calculates $t_2(n)$ on its "clock track." Next, M_D lets w be the maximal initial segment of doubled bits of x. Since the set $\{\, code(M) : M \text{ is a BM} \,\}$ is recursive, M_D can decide whether w is the *code*$'$ of a BM M in some time $U(n)$. The device of using w ensures that there are ∞-many inputs in which any given BM M is presented to M_D. If w is not a valid code, M_D halts and rejects.

If so, M_D runs M_U on input $code(M) \cdot code(x)$, except that after every pass by M_U, M_D calculates the μ-*time* of the pass and subtracts it from the total on its clock tape. If the total ever falls below $t_2(n)/2$, M_D halts and rejects. Otherwise, if the simulation of $M(x)$ finishes before the clock "rings," M_D rejects if M accepts, and accepts if M rejects. By Lemma 6.5, the total μ-*time* of M_D never exceeds $t_2(n)$.

Now let L be accepted by a BM M_L that runs in μ-*time* $t_1(n)$. Let K_1 be the constant overhead for M_U to simulate M_L in Theorem 6.3, and let K_2 be the overhead in Lemma 6.6.

Since t_1 is $o(t_2)$, there exists an x such that $t_2(|x|)/t_1(|x|) > 4K_1K_2$, the maximal initial segment $w \in (00 \cup 11)^*$ of x is $code'(M_L)$, and $U(|w|) < |x|$. Then the simulation of $M_L(x)$ by M_D finishes within μ-*time* $(1/2)t_2(|x|)$, and $M_D(x) \neq M_L(x)$. □

It is natural to ask whether the classes Dμ_dTIME$[t(n)]$ also form a tight hierarchy when t is held constant and d varies. The next section relates this to questions of determinism versus nondeterminism.

We observe finally that the BM in its original, reduced, and buffer forms all give the same definition of D$\mu_{\log}$TIME$[t(n)]$, and we have the following theorem.

THEOREM 6.8. *For any time functions* t_1, t_2 *such that* $t_1(n) \geq n$, $t_1 = o(t_2)$, *and* t_2 *is* $\mu_{\log}$-*time constructible,* D$\mu_{\log}$TIME$[t_1]$ *is properly contained in* D$\mu_{\log}$TIME$[t_2]$.

Proof. Here the strict boundary condition is not an issue, but the efficient universal simulation still requires delimiting the read block in advance. The idea is to locate cells $a_1, a_2, a_3, \ldots$, in the proof of Theorem 4.13 without addressing by the following trick. As in Theorem 4.14, the current address a_0 is already stored and $e = |a_0|$. In a rightward pull, rather than add $a_0 \dotplus e$, M' puts a_0 itself in binary rightward from cell a_0 on a separate track, appending an endmarker \$. By "recursively doubling" the string a_0, M' can likewise delimit the cells $a_2, a_3, \ldots$ Leftward pull steps are handled similarly, and put steps do not need $\mu_{\log}(a_0)$ at all. This is all that is needed for the efficient universal simulation. The remainder follows as above, since $\mu_{\log}$ is $\mu_{\log}$-*time* constructible—in fact, $\mu_{\log}(a) = |a|$ is computable in μ_1-*time* $O(|a|)$. □

A similar statement holds for the perhaps-larger $\mu_{\log}$-*time* classes for the BM variants that do use addressing.

7. Complexity theory and the BM model.

Our first result shows that the construction in the *Hennie–Stearns theorem* [33], which states that any multitape TM that runs in time $t(n)$ can be simulated by a two-tape TM in time $t(n)\log t(n)$, is memory efficient on the BM under μ_1. It has been observed in general that this construction is an efficient caching strategy. DTIME$[t(n)]$ refers to TM time, and DLIN stands for DTIME$[O(n)]$.

THEOREM 7.1. *For any function* t, DTIME$[t(n)] \subseteq$ Dμ_1TIME$[t(n)\log t(n)]$.

Proof. With reference to the treatment in [36], let M_1 be a multitape TM with alphabet Γ that runs in time $t(n)$, and let M_2 be the two-tape TM in the proof. The k-many tapes of M_1 are simulated on $2k$-many tracks of the first tape of M_2 so that all tape heads of M_1 are maintained on cell 0 of each track. M_2 uses its second tape only to transport blocks of the form $[2^{j-1} \ldots 2^j{-}1]$ from one part of the first tape to another. The functions used in these moves are homomorphisms between the alphabets Γ^{2k} and Γ^k that pack and unpack characters in blocks. Thus a BM M_3 simulating M_2 can compute each move in a single GST pass. By the structure of the blocks, any pass that incurs a memory-access charge of $\mu_1(2^j) = 2^j$ simulates at least 2^{j-1} moves of M_2. Hence the work and the μ_1 charges to M_3 are both $O(t(n)\log t(n))$. □

We do not know whether the random-access capability of a BM can be exploited to give an $O(t \log t)$ simulation that holds the work to $O(t)$, even for $\mu = \mu_{\log}$. Indeed, $O(t \log t)$ is the best bound we know for all memory-cost functions μ between $\mu_{\log}$ and μ_1. One consequence of this proposition is that sets in DLIN can be padded out to sets in TLIN.

COROLLARY 7.2. (a) *For every* $L \in$ DLIN, *the language* $\{\, x\#0^{|x|\log|x|} : x \in L \,\}$ *belongs to* TLIN.

(b) TLIN *contains* P-*complete languages, so* TLIN $\subseteq$ NC $\Longleftrightarrow$ P = NC.

Hence it is unlikely that all TLIN functions can be computed in polylog-many passes like the examples in this paper. If a BM quickly compresses the amount of information remaining to be processed into cells $[0 \ldots \sqrt{n}\,]$, it can then spend $O(\sqrt{n})$ time accessing these cells in any order desired and still run in linear μ_1-*time*.

THEOREM 7.3. *Let* M *be a BM that runs in* μ-*time* $t(n)$ *and space* $s(n)$. *Then we can find a DTM* T *that simulates* M *in time* $O[t(n)s(n)/\mu(s(n))]$.

Proof. T has two tapes, one for the main tape of M and one used as temporary storage for the output in passes. (If M has the buffer mechanism, then the second tape of T simulates the buffer.) Let s stand for $s(n)$. Consider a move by M that changes the current address a to $\lfloor a/2 \rfloor$. T can find this cell in at most $3a/2$ steps by keeping count with its second tape. Since $s/a \geq 1$, the tracking property $\mu(Na) \leq N\mu(a)$ with $N := s/a$ gives $a/\mu(a) \leq s/\mu(s)$. Hence the ratio of the time used by T to the μ-time charged to M stays $O[s/\mu(s)]$. The same holds for the moves $a := 2a$ and $a := 2a+1$. T has every GST S of M in its finite control, and simulates a pull by writing $S[a \dots b]$ to its second tape, moving to cell 0, copying $S[a \dots b]$ over the first tape, and moving back to cell a. Both this and the analogous simulation of a put by T take time $O(a + b)$, and even the ratio of this to the memory-access charges $\mu(a) + \mu(b)$, not even counting the number of bits processed by M, keeps the running total of the time logged by T below $t(n)s/\mu(s)$. □

COROLLARY 7.4. *For any time bound* $t(n) \geq n$, $\mathrm{D}\mu_1\mathrm{TIME}[t(n)] \subseteq \mathrm{DTIME}[t(n)]$. *In particular,* TLIN $\subseteq$ DLIN.

More generally, for any $d \geq 1$, $\mathrm{D}\mu_d\mathrm{TIME}[t(n)] \subseteq \mathrm{DTIME}[t^{2-(1/d)}(n)]$. Allowing TMs to have d-dimensional tapes brings this back to a linear-time simulation.

LEMMA 7.5. *For any integer* $d \geq 1$ *and time bound* $t(n) \geq n$, *a BM* M *that runs in* μ_d*-time* $t(n)$ *can be simulated in time* $O(t(n))$ *by a* d*-dimensional TM* T.

Proof. T has one d-dimensional tape on which it winds the main tape of M in a spiral about the origin, and one linear tape on which it buffers outputs by the GST S of M. In any pass that incurs a μ_d charge of $a^{1/d}$, T can walk between cell a and the origin within $a^{1/d}$ steps and complete the move. □

Let us say that a language or function in $\mathrm{D}\mu_d\mathrm{TIME}[O(n)]$ *has dimension* d. For a problem above linear time, we could say that its *dimensionality* is the least d, if any, for which the problem has relatively optimal BM programs that are μ_d-efficient (see Definition 2.10). The main robustness theorem is our justification for this concept of dimensionality. Lemma 7.5 says that it is no less restrictive than the older concept given by d-dimensional Turing machines. For $d > 1$ we suspect that it is noticeably more restrictive. The d-dimensional tape reduction theorem of Paul, Seiferas, and Simon [58] gives $t'(n)$ roughly equal to $t(n)^{1+1/d}$, and when ported to a BM, incurs memory access charges close to $t(n)^{1+2/d}$. Intuitively, the problem is that a d-dimensional TM can change the direction of motion of its tape head(s) at any step, whereas this would be considered a break in pipelining for the simulating BM, and thus subject to a memory-access charge.

We write $\mathrm{RAM\text{-}TIME}^{\log}$ for time on the log-cost RAM. A log-cost RAM can be simulated with constant-factor overhead by a TM with one binary tree-structured tape and one standard worktape [57], and the latter is simulated in real time by a RAM-TM.

PROPOSITION 7.6. *For any time function* t,

(a) $\mathrm{RAM\text{-}TIME}^{\log}[t(n)] \subseteq \mathrm{D}\mu_{\log}\mathrm{TIME}[t(n)\log t(n)]$.

(b) $\mathrm{D}\mu_{\log}\mathrm{TIME}[t(n)] \subseteq \mathrm{RAM\text{-}TIME}^{\log}[t(n)\log t(n)]$.

Proof. Straightforward simulations give these bounds. (The extra $\log t(n)$ factor in (b) dominates a factor of $\log\log n$ that was observed by [44] for the simulation of a TM (or RAM-TM) by a log-cost RAM.) □

For *quasilinear time*, i.e., time $qlin = n(\log n)^{O(1)}$, the extra $\log n$ factors in Theorem 7.1 and Proposition 7.6 do not matter. Following Schnorr [65], we write DQL and NQL for the TM time classes DTIME[$qlin$] and NTIME[$qlin$]. Gurevich and Shelah [30] proved that $\mathrm{RAM\text{-}TIME}^{\log}[qlin]$ is the same as deterministic nearly linear time on the RAM-TM and several other RAM-like models, and perhaps more surprisingly, that the nondeterministic counterparts of these classes are all equal to NQL.

COROLLARY 7.7. (a) $\mathrm{D}\mu_1\mathrm{TIME}[qlin] = \mathrm{DQL}$.

(b) $\mathrm{D}\mu_{\log}\mathrm{TIME}[qlin] = \mathrm{RAM\text{-}TIME}^{\log}[qlin] \subseteq \mathrm{NQL}$.

Hence the objective of separating the classes DμTIME[$O(n)$], as μ varies from μ_1 through μ_d to $\mu_{\log}$, by anything more than factors of $O(\log n)$, runs into the problem of whether DQL $\neq$ NQL, which seems as hard as showing P $\neq$ NP. Whether they can be separated by even one $\log n$ factor is discussed in the next section.

8. Open problems and further research. The following languages have been much studied in connection with linear-time algorithms and nonlinear lower bounds. We suppose that the lists in L_{dup} and L_{int} are all normal.

(a) Pattern matching: $L_{pat} = \{\, p\#t : (\exists u, v \in \{0,1\}^*)\, t = upv \,\}$.

(b) Element (non)distinctness: $L_{dup} = \{\, x_1\#\cdots\#x_m : (\exists i, j)\, i < j \wedge x_i = x_j \,\}$.

(c) List intersection: $L_{int} = \{\, x_1\#\cdots\#x_m, y_1\#\cdots\#y_m : (\exists i, j)\, x_i = y_j \,\}$.

(d) Triangle: $L_{\Delta} = \{A : A$ is the adjacency matrix of an undirected graph that contains a triangle$\}$.

L_{pat} belongs to DLIN (see [25], [23]), and was recently shown not to be solvable by a one-way non-sensing multihead DFA [42]. L_{dup} and L_{int} can be solved in linear time by a RAM or RAM-TM that treats list elements as cell addresses. L_{Δ} is not believed to be solvable in linear time on a RAM at all. The best method known involves computing $A^2 + A$, and squaring $n \times n$ integer matrices takes time approximately $N^{1.188}$, where $N = n^2$, by the methods of [19]. (For directed triangles, cubing A is the best way known.)

OPEN PROBLEM 1. *Do any of the above languages belong to* TLIN? *If not, prove nonlinear lower bounds.*

A BM can be made nondeterministic (NBM) by letting $\delta(q, c)$ be multiply valued and, more strongly, by using nondeterministic GSTs or GSM mappings in block moves. Define NTLIN to be linear time for NBMs of the *weaker* kind. Then all four of the above languages belong to NTLIN. Moreover, they require only $O(\log n)$ bits of nondeterminism.

OPEN PROBLEM 2. *Is* NTLIN $\neq$ TLIN? *For reasonable μ and time bounds t, is there a general separation of* NμTIME[$t(n)$] *from* DμTIME[$t(n)$]?

Grandjean [27], [28] shows that a few NP-complete languages are also hard for NLIN under TM linear time reductions, and hence by the theorem of [56] lie outside DLIN, not to mention TLIN. However, these languages seem not to belong to NTLIN, nor even to linear time for NBMs of the stronger kind. The main robustness theorem and subsequent simulations hold for the weaker kind of nondeterminism, but our proofs do not work for the stronger because they rerun the GST S used in a pass. We suspect that different proofs will give similar results. A separation of the two kinds can be shown with regard to the pass-count measure $R(n)$, which serves as a measure of parallel time (e.g., $R(n) = \text{polylog}(n)$ and polynomial work $w(n)$ by deterministic BMs characterizes NC [62]). P. van Emde Boas [personal communication, 1994] has observed that while deterministic BMs and NBMs of the weaker kind belong to the *second machine class* of [68] with $R(n)$ as time measure, NBMs of the stronger kind have properties shown there to place models beyond the second machine class. Related to Open Problem 2 is whether the classes Dμ_dTIME[$O(n)$] differ as d varies. It is also natural to study *memory-efficient* reductions among problems.

The following idea for obtaining such separations and proving nonlinear lower bounds in *μ-time* on a deterministic BM M suggests itself: let $\Gamma_{M,x}$ stand for the set of access points used in the computation of the BM M on input x. In order for M to run in linear *μ-time*, $\Gamma_{M,x}$ must thin out at the high end of memory. In particular for $\mu = \mu_1$, there are long segments between access points that can be visited only a constant number of times. The technical difficulty is that block moves can still transport information processed in low memory to these segments, and the proof of Theorem 7.1 suggests that a lower bound of $\Omega[n \log n]$ may be the best achievable in this manner. In general, we advance the BM as a logical next step in the longstanding program of proving nonlinear lower bounds for natural models of computation.

In particular, we ask whether the techniques used by Dietzfelbinger, Maass, and Schnitger [20] to obtain lower bounds for Boolean matrix transpose and several sorting-related functions on a certain restricted two-tape TM can be applied to the differently restricted kind of two-tape TM in Theorems 7.1 and 7.3. The latter kind is equivalent to a TM with one worktape and one pushdown store with the restriction that after any POP, the entire store must be emptied before the next PUSH.

We have found two variants to the BM model that seem to depart from the cluster of robustness results shown in this paper. They relate to generally known issues of *delay* in computations. The first definition is the special case for GSTs of Manacher's notion of a "fractional on-line RAM algorithm with steady-paced output" [53].

DEFINITION 8.1. *Let $d \geq 0$ and $e \geq 1$ be integers. A GST S runs in* fixed output delay d/e *if for every terminal trajectory $(q_0, x_0, q_1, \ldots, x_{m-1}, q_m)$, and each $i \leq m-2$, $|\rho(q_i, x_i)| = d$ if e divides $i+1$, $= 0$ otherwise. For the exiting transition, $|\rho(q_{m-1}, x_{m-1})|$ depends only on $(m \bmod e)$. The quantity $C := d/e$ is called the* expansion factor *of S.*

Note that the case $d = 0$ is allowed. Every GST function g can be written as $e \circ f$, where f is fixed delay and e is an erasing homomorphism: pad each output of the GST for g to the same length with "@" symbols, and let e erase them. A k-input GST *with stationary moves allowed* may keep any of its input heads stationary in a transition. Such a machine can be converted to an equivalent form coded like an ordinary GST in which every state q has a label $j \in \{1, \ldots, k\}$ such that q reads and advances only the head on tape j.

DEFINITION 8.2. (a) *A BM* runs in fixed output delay *if every GST chip in M runs in fixed output delay.*

(b) *A* pause buffer BM *is a BM with buffer whose put steps may use 2-input GSTs with variable input delay (cf. Proposition* 4.3*).*

Put another way, the BM model presented in this paper requires fixed delay in reading input but not in writing output, while (a) requires both and (b) requires neither. We did not adopt (b) because we feel that stationary moves by a 2-GST in the course of a pass require communication between the heads, insofar as further movements depend on the current input symbols, and hence should incur memory-access charges. We defend our choice against a similar criticism that would require (a) by contending that in a single-tape GST pass, the motion of the read head is not affected by the write head, and the motion of the write head depends only on local factors as bits come in to it. Also, every BM has a limit C on the number of output bits per input bit read by a GST. The main robustness theorem (Theorem 3.1), in particular the ability to forecast the length of the output of a pass by fixed-delay means shown in Theorem 4.13, satisfies our doubts about this.

The robustness results in this paper do carry over to the case of fixed output delay.

THEOREM 8.1. *For any rational $d \geq 1$, the fixed-delay restrictions of the BM and all the variants defined in* §3 *simulate each other up to constant factors in μ_d-time.*

Proof. All auxiliary operations in the simulations in §4 use GSTs that run in fixed output delay, except for the second, unpadded run of the GST S in Theorem 4.13. However, if S already runs in fixed output delay, so does this run. □

Under the proof of Theorem 2.1, the corresponding notion for the reduced form of the model is "fixed delay after the initial transition." Our proof of efficient universal simulation does not quite carry over for fixed output delay because the quantities k and l in the proof of Theorem 6.3 may differ for different M. The operations that pull off the word g_z and the padded output *code*(y) run in "stride" a function of k and l, but this is not fixed. We believe that the proof can be modified to do so under a different representation scheme for monoids.

Whether a similar robustness theorem holds for the pause-buffer BM leads to an open problem of independent interest: can every k-input GST be simulated by a composition tree of two-input GSTs when stationary moves are allowed? The questions of the power of both variants versus the basic BM can be put in concrete terms.

OPEN PROBLEM 3. *Can the homomorphism $Er_2 : \{0, 1, 2\}^* \to \{0, 1\}^*$, which erases all 2's in its argument, be computed in linear μ_1-time by a BM that runs in fixed output delay?*

OPEN PROBLEM 4. *For every two-input GST S with stationary moves allowed, does the function $S'(x\#y) := S(x, y)$ belong to* TLIN?

THEOREM 8.2. (a) *The answer to Open Problem* 3 *is "yes" iff for every memory-cost function μ and BM M, there is a BM M' that runs in fixed output delay and simulates M linearly under μ.*

(b) *The answer to Open Problem* 4 *is "yes" iff for every memory-cost function μ and pause-buffer BM M, there is a BM M' that simulates M linearly under μ.*

Proof. For the forward implication of (a), M' pads every output by M with @ symbols, coding the rest over $\{0, 1\}^*$, and runs $Er_@$ on a separate track to remove the padding. That of (b) is proved along the lines of Proposition 4.3. The reverse implications are immediate, and all this needs only the tracking property of μ. □

Alon and Maass [4] prove substantial time–space tradeoffs for the related "sequence-equality" problem SE[n]: given $x, y \in \{0, 1, 2\}^n$, does $Er_2(x) = Er_2(y)$? We inquire whether their techniques, or those of [54], can be adapted to the BM. The BM in Theorem 7.1 runs in output delay 1/2, 1, or 2 for all passes, so the two kinds of BM can be separated by no more than a log factor. A related question is whether every language in TLIN, with or without fixed output delay, has linear-sized circuits.

Further avenues for research include analyzing implementations of certain important algorithms on the BM, as done for the BT and UMH in [2], [5]. Here the BM is helped by its proximity to the Pratt–Stockmeyer vector machine, since conserving memory-access charges and parallel time often lead to similar methods. One can also study storage that is expressly laid out on a two-dimensional grid or in three-dimensional space, where a *pass* might be defined to follow either a one-dimensional line or a two-dimensional plane. We expect the former not to be much different from the BM model with its one-dimensional tape, and we also note that CD-ROM and several current two-dimensional drive technologies use long one-dimensional tracks. The important issue may not be so much the topology of the memory itself, but whether "locality is one-dimensional" for purposes of pipelining.

Last, we ask about meaningful technical improvements to the simulations in this paper. The lone obstacle to extending the main robustness theorem for $\mu = \mu_{\log}$ is the simulation of random access by tree access in Lemma 4.6. The constants on our universal simulation are fairly large, and we seek a more efficient way of representing monoids and computing the products. Two more questions are whether the BM loses power if the move option $a := 2a+1$ is eliminated, and whether the number m of markers in a finger BM can be reduced to $m-1$ or to 4 without multiplying the number of block moves by a factor of $\log t(n)$.

9. Conclusion. In common with motivations expressed in [2] and [5], the BM model fosters a finer analysis of many theoretical algorithms in terms of how they use memory and how they really behave in running time when certain practicalities of implementation are taken into account. We have shown that the BM model is quite robust and that the concept of functions and languages being computable in a memory-efficient manner does not depend on technical details of setting up the model. The richer forms of the model are fairly natural to program, providing random access and the convenience of regarding finite transductions such as addition and vector Booleans as basic operations. The tightest form of the model is syntactically simple, retains the bit-string concreteness of the TM, and seems to be a tractable object of study for lower bound arguments. The robustness is evidence that our abstraction is "right."

In contrast to the extensive study of polynomial-time computation, very little is known about linear-time computation. Owing to an apparent lack of linear-time robustness among various kinds of TMs, RAMs, and other machines, several authorities have queried their suitability as a model for computation in $O(n)$ time. Since we have μ as a parameter we have admittedly not given a single answer to the question, "What is linear time?", and leave TLIN, Dμ_2TIME[$O(n)$], and Dμ_3TIME[$O(n)$] as leading candidates. However, the BM model does supply a robust yardstick for assessing the complexity of many natural combinatorial problems and for investigating the structure of several other linear-time complexity classes. It has a tight deterministic time hierarchy right down to linear time. The efficient universal simulator which we have constructed to show this result uses the word problem for finite monoids in an interesting manner. The longstanding program of showing nonlinear lower bounds in reasonable models of computation has progressed up to machines apparently just below the BM (under μ_1) in power, so that attacking the problems given here seems a logical next step. The authors of [3] refer to the "challenging open problem" of extending their results when bit manipulations for dissecting records are available. The bit operations given to the BM seem to be an appropriate setting for this problem. A true measure of the usefulness of the BM model will be whether it provides good ground for developing and connecting methods that solve older problems not framed with the term "BM." We offer the technical content of this paper as appropriately diligent spadework.

Appendix: Proof of Theorem 2.1. For every move state q in M we add a new GST S_q that performs a 1-bit empty pull just to read the currently scanned character d, and then sends control to $\delta(q, d)$. This modification no more than doubles the μ access charges, and gives M the following property: for any pass by a GST S_i, the next GST S_k to be invoked (or HALT) is a function only of i and the character c that caused S_i to exit, and there is at most one intervening move. Henceforth we assume that M has this form, and number its GST chips by $S_0, \ldots, S_r$, with S_0 as start chip.

M' uses an alphabet Γ' which includes the alphabet Γ of M, a surrogate blank @, tokens $\{ s_0, \ldots, s_r \}$ for the chips of M, markers $\{ m_U, m_L, m_R, m_{no}, m_H \}$ for the three kinds of move, "no move," and HALT, special *instruction markers* $\{ I_0, \ldots, I_{12} \}$, *plus* certain tuples of length up to 7 of the foregoing characters. We also use @ to indicate that the symbol written to cell 0 is immaterial.

During the simulation, the first component of every tuple in a cell i is the character $c_i \in \Gamma$ in that cell of the tape of M. Except initially, cell 1 holds both c_0 and c_1, so that cell 0 can be overwritten by other characters. This also allows M' to simulate all moves by M without ever moving its own cell-a head back to cell 0. The markers I_0 and I_1 tell M' when the cell-a head of M is in cell 0 or 1. For $a \geq 2$, the heads of M and M' coincide. The other main invariant of the simulation is that the only cell besides cells 0 and 1 to contain multiple symbols is cell a. The two initial moves of M' set up these invariants.

Character(s) read	Action (Initial mode is Ra, $a = 0$.)
c_0, c_1	Pull $[c_0, c_1]$ to cell 0, $a := a$, *mode* $:= 0R$.
$[c_0, c_1], c_1$	Put @ into cell 0 and $[c_1, c_0, s_0, I_0]$ into cell 1, $a := 2a + 1$, *mode* $:= Ra$.

The first move must automatically be executed every time M' moves its tape head to a new cell a, $a \geq 2$, since this cell and cell $a + 1$ will always contain single characters over Γ. However, the second move is unique to the initialization because cell 1 will never again hold a single character. The cell-a head of M' is now on cell 1, but the I_0 enables M to record that the cell-a head of M is still on cell 0.

The lone GST S of M' includes two copies of each GST S_i of M. The first is a "dummy copy" which simulates S_i but suppresses output until it picks up the character c that causes S_i to exit. On this exiting transition, the dummy outputs a token s_k for the next GST S_k and a token m for the intervening move, or m_{no} for none, or m_H for HALT. The other copy simulates the actual pass by S_i. It has special states that distinguish whether S_i has written zero, one, or at least two output symbols in the pass, since the first one or two symbols of the output y are altered. If S_i performs a pull and $|y| \geq 2$, we define $c'_0 := y_0$ if $y_0 \neq B$, but $c'_0 := c_0$ if $y_0 = B$. Similarly $c'_1 := y_1$ if $y_1 \neq B$, but $c'_1 := c_1$ if $y_1 = B$. On the tape of M', the output y looks like $[c'_0, c'_1, \ldots][c'_1, c'_0, \ldots]y_2 \cdots y_l$, where $l = |y|$. For $|y| \leq 1$, treat the missing y_1 and/or y_0 as B. Besides these functional conventions on s_k, m, c'_0, and c'_1, we omit reference to the address a if it is not changed and omit the second character read by S when it does not affect control at the large initial branch. Let S_i be the current GST of M.

Character(s) read	Action (Current mode is Ra, $a = 1$.)
$[c_1, c_0, s_i, I_0]$	By the validity conditions (Definition 2.3), the output y by S_i has length at most 2. Hence the next-move token m and next-GST token s_k can be picked up and the output y written in one pass, without needing the dummy copy of S_i. If $m = m_H$, S pulls @ to cell 0 and $[c'_1, c'_0, I_{12}]$ to cell 1. If $m = m_R$, pulls $@[c'_1, c'_0, s_k, I_1]$ to signify that the cell-a head of M is now on cell 1. Else S pulls $@[c'_1, c'_0, s_k, I_0]$, and this step repeats. In each case, *mode* $:= Ra$.
$[c_1, c_0, I_{12}]$	Pull c_0 into cell 0, c_1 into cell 1, and HALT.
$[c_1, c_0, s_i, I_1]$	Simulate S_i as for $[c_1, c_0, s_i, I_0]$ to get m, s_k, and y, but treat c_1 as the first input character to S_i. If $m = m_H$ pull $@[c'_1, c'_0, I_{12}]$, if $m = m_U$ pull $@[c'_1, c'_0, s_k, I_0]$, and if $m = m_{no}$, pull $@[c'_1, c'_0, s_k, I_1]$. In these three cases, the address of M' stays at 1. If $m = m_L$, then pull $@[c'_1, c'_0, s_k]$ and effect $a := 2a$. If $m = m_R$, pull $@[c'_1, c'_0, s_k]$ and effect $a := 2a + 1$. In every case, the mode stays Ra.

The last two cases give $a \geq 2$. When $a \geq 2$, the next pass by S encounters a single character $c_a \in \Gamma$ on its start transition (possibly $c_a = B$), and S must perform the first operation above. This overwrites the @ in cell 0. However, the new character $[c'_1, c'_0, s_k]$ in cell 1 prevents the initial sequence from recurring, viz. the following:

Character(s) read	Action (Current mode is Ra, $a \geq 2$.)
c_a, c_{a+1}	Pull $[c_a, c_{a+1}]$ to cell 0, $a := a$, *mode* $:= 0R$.
B	Pull $[@, @, I_0]$ to cell 0, $a := a$, *mode* $:= 0R$.
$[c_a, c_{a+1}]$, $[c_1, c_0, s_i]$	Put $[c_a, c_0, c_1, s_i, I_2]$ into cell a. If the label of S_i is La then *mode* $:= La$, else *mode* $:= Ra$.
$[c_a, c_0, c_1, s_i, I_2]$	If S_i is labeled $0L$ or $0R$, then pull $[c_0, c_1, c_a, s_i, I_6]$ into cell 0, and *mode* $:=$ the mode of S_i. Else S simulates the dummy copy of S_i to find m and s_k, treating c_a as the first input character to S_i, and pulls $[c_0, c_1, c_a, m, s_k, s_i, I_3]$ to cell 0 with *mode* $:= 0R$.
$[c_0, c_1, c_a, m, s_k, s_i, I_3]$	Put $[c_a, c_0, c_1, m, s_k, s_i, I_4]$ into cell a, *mode* $:=$ the mode of S_i.
$[c_a, c_0, c_1, m, s_k, s_i, I_4]$	Simulate the pull move by S_i, translating its output y to $[c'_0, c'_1, c_a, m, I_5][c'_1, c'_0, s_k]y_2 \cdots y_l$, and change *mode* to $0R$. *Remark:* For c_a to be correct, it is vital that cell a not be overwritten in this pull.

$[c_0, c_1, c_a, m, I_5]$ Put c_a into cell a. On exit, if $m = m_{no}$ then leave a unchanged, if $m = m_U$ effect $a := \lfloor a/2 \rfloor$, if $m = m_L$ effect $a := 2a$, and if $m = m_R$ effect $a := 2a + 1$. In each of these four cases, *mode* $:= Ra$. For $m = m_H$, see below.

If the last move was *up*, i.e., a to $\lfloor a/2 \rfloor$, we may now have $a = 1$ again. Since the "sentinel" in cell 1 is always correctly updated to the next GST S_i, this is handled by the following:

$[c_1, c_0, s_i]$ Same as for $[c_1, c_0, s_i, I_1]$.

If still $a \geq 2$, then S once again senses single characters in cells a and $a + 1$, and the cycle repeats. The other branch with instruction 6 goes as follows:

$[c_0, c_1, c_a, s_i, I_6]$ Here S_i is labeled $0L$ or $0R$, and this is the current mode. S treats c_0, c_1 as the first two input characters in simulating the dummy copy of S_i, and puts $[c_a, c_0, c_1, m, s_k, s_i, I_7]$ into cell a with *mode* $:= Ra$.

$[c_a, c_0, c_1, m, s_k, s_i, I_7]$ Pull $[c_0, c_1, c_a, m, s_k, s_i, I_8]$ into cell 0, *mode* := the mode of S_i.

$[c_0, c_1, c_a, m, s_k, s_i, I_8]$ Simulate the put by S_i. If the output y is empty or begins with B, let $c'_a := c_a$. Else let $c'_a := y_0$. Copy y as $[c'_a, c_0, c_1, m, s_k, I_9]$, and set *mode* $:= 0R$.

$[c_a, c_0, c_1, m, s_k, I_9]$ Pull $[c_0, c_1, c_a, m, I_5]$ to cell 0 and $[c_1, c_0, s_k]$ to cell 1, *mode* $:= Ra$.

The validity conditions prevent cell a from being overwritten in a pull. It is possible for cell 1 to be overwritten by a leftward put that exits after just one input bit, but this can only happen if $a \leq C$, where C is the maximum number of bits a leftward pull chip of M can write in its first transition. The problem can be solved either by exploiting the ability of M itself to remember C-many characters in its finite control, or by reprogramming M so that no leftward pull chip outputs more than one symbol on its first step. Details are left to the reader.

The final halting routine involves a "staircase" to leave the tape exactly the same as that of M at the end of the computation. It picks up in the case $[c_0, c_1, c_a, m, I_5]$ with $m = m_H$.

$[c_0, c_1, c_a, m_H, I_5]$ Put $[c_a, c_0, c_1, I_{10}]$ into cell a, *mode* $:= Ra$.

$[c_a, c_0, c_1, I_{10}]$ Pull $[c_0, c_1, c_a, I_{11}]$ to cell 0 and $[c_1, c_0, I_{12}]$ to cell 1, with *mode* $:= 0R$.

$[c_0, c_1, c_a, I_{11}]$ Put c_a into cell a, effect $a := \lfloor a/2 \rfloor$, *mode* $:= Ra$.

c_a, c_{a+1} Pull $[c_a, c_{a+1}]$ to cell 0, *mode* $:= 0R$.

$[c_a, c_{a+1}], [c_1, c_0, I_{12}]$ Put c_a into cell a, effect $a := \lfloor a/2 \rfloor$, *mode* $:= Ra$.

$[c_1, c_0, I_{12}]$ As above, pull c_0 into cell 0, c_1 into cell 1, and HALT.

M' uses exactly the same tape cells as M, making at most eight passes of equal or less cost for each pass by M. The final "staircase" down from cell a is accounted against the μ-charges for M to have moved out to cell a. Hence both the number of bits processed by M' and the μ-*acc* charges to M' are within a constant factor of their counterparts in M.

For the converse simulation of the reduced form S by a BM M, the only technical difficulty is that S may have different exiting transitions on the same character c. The solution is to run a dummy copy of S that outputs a token t for the state in which S exits. Then t is used to send control to the move state of M' that corresponds to the label $l_1(t)$, and thence to a copy of S with the pass-type label $l_2(t)$. The details of running the dummy copy are the same as above. □

By using more "instruction markers" one can make the mode of M' always follow the cycle $Ra, 0R, La, 0L$. Hence the only decision that need depend on the terminal state of the lone GST S is the next origin cell a.

Acknowledgments. I would like to thank Professor Michael C. Loui for comments on preliminary drafts of this work and for bringing [2] and [5] to my attention. Professors Pierre McKenzie and Gilles Brassard gave me a helpful early opportunity to test these ideas at an open forum in a University of Montreal seminar. Special thanks are due to Professors Klaus Ambos-Spies, Steven Homer, Uwe Schöning, and other organizers of the 1992 Schloss Dagstuhl Workshop on Structural Complexity for inviting me to give a presentation out of which the present work grew. Last, I thank an anonymous referee and several colleagues for helpful suggestions on nomenclature and presentation.

REFERENCES

[1] A. Aggarwal, B. Alpern, A. Chandra, and M. Snir, *A model for hierarchical memory*, in Proc. 19th Symposium on the Theory of Computing, Association for Computing Machinery, New York, 1987, pp. 305–314.

[2] A. Aggarwal, A. Chandra, and M. Snir, *Hierarchical memory with block transfer*, in Proc. 28th Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1987, pp. 204–216.

[3] A. Aggarwal and J. Vitter, *The input-output complexity of sorting and related problems*, Comm. Assoc. Comput. Mach., 31 (1988), pp. 1116–1127.

[4] N. Alon and W. Maass, *Meanders and their application to lower bound arguments*, J. Comput. System Sci., 37 (1988), pp. 118–129.

[5] B. Alpern, L. Carter, and E. Feig, *Uniform memory hierarchies*, in Proc. 31st Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1990, pp. 600–608.

[6] J. Balcázar, J. Díaz, and J. Gabarró, *Structural Complexity Theory*, Springer-Verlag, Berlin, New York, 1988.

[7] D. M. Barrington, N. Immerman, and H. Straubing, *On uniformity within* NC^1, in Proc. 3rd Structures, IEEE Computer Society Press, Los Alamitos, CA, 1988, pp. 47–59.

[8] ———, *On uniformity within* NC^1, J. Comput. System Sci., 41 (1990), pp. 274–306.

[9] A. Ben-Amram and Z. Galil, *Lower bounds for data structure problems on RAMs*, in Proc. 32nd Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1991, pp. 622–631.

[10] ———, *On pointers versus addresses*, J. Assoc. Comput. Mach., 39 (1992), pp. 617–648.

[11] G. Blelloch, *Vector Models for Data-Parallel Computing*, MIT Press, Cambridge, MA, 1990.

[12] M. Blum, *A machine-independent theory of the complexity of recursive functions*, J. Assoc. Comput. Mach., 14 (1967), pp. 322–336.

[13] R. Book and S. Greibach, *Quasi-realtime languages*, Math. Systems Theory, 4 (1970), pp. 97–111.

[14] A. Chandra, D. Kozen, and L. Stockmeyer, *Alternation*, J. Assoc. Comput. Mach., 28 (1981), pp. 114–133.

[15] J. Chang, O. Ibarra, and A. Vergis, *On the power of one-way communication*, J. Assoc. Comput. Mach., 35 (1988), pp. 697–726.

[16] J. Chen and C. Yap, *Reversal complexity*, SIAM J. Comput., 20 (1991), pp. 622–638.

[17] M. Conner, *Sequential machines realized by group representations*, Inform. and Comp., 85 (1990), pp. 183–201.

[18] S. Cook and R. Reckhow, *Time bounded random access machines*, J. Comput. System Sci., 7 (1973), pp. 354–375.

[19] D. Coppersmith and S. Winograd, *Matrix multiplication via arithmetical progressions*, J. Symbolic Comput., 9 (1990), pp. 251–280.

[20] M. Dietzfelbinger, W. Maass, and G. Schnitger, *The complexity of matrix transposition on one-tape off-line Turing machines*, Theoret. Comput. Sci., 82 (1991), pp. 113–129.

[21] C. Elgot and A. Robinson, *Random-access stored-program machines*, J. Assoc. Comput. Mach., 11 (1964), pp. 365–399.

[22] Y. Feldman and E. Shapiro, *Spatial machines: A more-realistic approach to parallel computation*, Comm. Assoc. Comput. Mach., 35 (1992), pp. 60–73.

[23] P. Fischer, A. Meyer, and A. Rosenberg, *Real-time simulations of multihead tape units*, J. Assoc. Comput. Mach., 19 (1972), pp. 590–607.

[24] M. Fürer, *Data structures for distributed counting*, J. Comput. System Sci., 29 (1984), pp. 231–243.

[25] Z. Galil and J. Seiferas, *Time-space optimal string matching*, J. Comput. System Sci., 26 (1983), pp. 280–294.

[26] E. Graedel, *On the notion of linear-time computability*, Internat. J. Found. Comput. Sci., 1 (1990), pp. 295–307.

[27] E. Grandjean, *A natural NP-complete problem with a nontrivial lower bound*, SIAM J. Comput., 17 (1988), pp. 786–809.

[28] ———, *A nontrivial lower bound for an NP problem on automata*, SIAM J. Comput., 19 (1990), pp. 438–451.

[29] E. Grandjean and J. Robson, *RAM with compact memory: A robust and realistic model of computation*, in Proc. 4th Annual Workshop in Computer Science Logic, Lecture Notes in Comput. Sci., 533, (1991), pp. 195–233.

[30] Y. Gurevich and S. Shelah, *Nearly linear time*, in Proc. Logic at Botik, Lecture Notes in Comput. Sci., 363 (1989), pp. 108–118.

[31] J. Hartmanis and R. Stearns, *On the computational complexity of algorithms*, Trans. Amer. Math. Soc., 117 (1965), pp. 285–306.

[32] ———, *Algebraic Structure Theory of Sequential Machines*, Prentice–Hall, Englewood Cliffs, NJ, 1966.

[33] F. Hennie and R. Stearns, *Two-way simulation of multitape Turing machines*, J. Assoc. Comput. Mach., 13 (1966), pp. 533–546.

[34] T. Heywood and S. Ranka, *A practical hierarchical model of parallel computation* I: *The model*, J. Parallel Distrib. Comput., 16 (1992), pp. 212–232.

[35] J.-W. Hong, *On similarity and duality of computation* I, Inform. and Comp., 62 (1985), pp. 109–128.

[36] J. Hopcroft and J. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison–Wesley, Reading, MA, 1979.

[37] O. Ibarra, *Systolic arrays: Characterizations and complexity*, in Proc. 1986 Conference on Mathematical Foundations of Computer Science, Lecture Notes in Comput. Science, 233 (1986), pp. 140–153.

[38] O. Ibarra and T. Jiang, *On one-way cellular arrays*, SIAM J. Comput., 16 (1987), pp. 1135–1153.

[39] O. Ibarra and S. Kim, *Characterizations and computational complexity of systolic trellis automata*, Theoret. Comput. Sci., 29 (1984), pp. 123–153.

[40] O. Ibarra, S. Kim, and M. Palis, *Designing systolic algorithms using sequential machines*, IEEE Trans. Comput., 35 (1986), pp. 531–542.

[41] O. Ibarra, M. Palis, and S. Kim, *Some results concerning linear iterative (systolic) arrays*, J. Par. Dist. Comp., 2 (1985), pp. 182–218.

[42] T. Jiang and M. Li, *K one-way heads cannot do string-matching*, in Proc. 25th Symposium on the Theory of Computing, Association for Computing Machinery, New York, 1993, pp. 62–70.

[43] T. Kameda and R. Vollmar, *Note on tape reversal complexity of languages*, Inform. and Control, 17 (1970), pp. 203–215.

[44] J. Katajainen, J. van Leeuwen, and M. Penttonen, *Fast simulation of Turing machines by random access machines*, SIAM J. Comput., 17 (1988), pp. 77–88.

[45] A. Kolmogorov and V. Uspenskii, *On the definition of an algorithm*, Uspekhi Mat. Nauk, 13 (1958), pp. 3–28; English translation, Russian Math. Surveys, 30 (1963), pp. 217–245.

[46] R. Kosaraju, *Real-time simulation of concatenable double-ended queues by double-ended queues*, in Proc. 11th Symposium on the Theory of Computing, 1979, pp. 346–351.

[47] K. Krohn and J. Rhodes, *Algebraic theory of machines* I: *Prime decomposition theorem for finite semigroups and machines*, Trans. Amer. Math. Soc., 116 (1965), pp. 450–464.

[48] K. Krohn, J. Rhodes, and B. Tilson, *The prime decomposition theorem of the algebraic theory of machines*, in Algebraic Theory of Machines, Languages, and Semigroups, M. Arbib, Ed., Academic Press, New York, 1968, Chap. 4–9.

[49] D. Leivant, *Descriptive characterizations of computational complexity*, J. Comput. System Sci., 39 (1989), pp. 51–83.

[50] M. Loui, *Simulations among multidimensional Turing machines*, Theoret. Comput. Sci., 21 (1981), pp. 145–161.

[51] ———, *Optimal dynamic embedding of trees into arrays*, SIAM J. Comput., 12 (1983), pp. 463–472.

[52] ———, *Minimizing access pointers into trees and arrays*, J. Comput. System Sci., 28 (1984), pp. 359–378.

[53] G. Manacher, *Steady-paced-output and fractional-on-line algorithms on a RAM*, Inform. Process. Lett., 15 (1982), pp. 47–52.

[54] Y. Mansour, N. Nisan, and P. Tiwari, *The computational complexity of universal hashing*, Theoret. Comput. Sci., 107 (1993), pp. 121–133.

[55] D. Muller and F. Preparata, *Bounds to complexities of networks for sorting and switching*, J. Assoc. Comput. Mach., 22 (1975), pp. 195–201.

[56] W. Paul, N. Pippenger, E. Szemerédi, and W. Trotter, *On determinism versus nondeterminism and related problems*, in Proc. 24th Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1983, pp. 429–438.

[57] W. Paul and R. Reischuk, *On time versus space* II, J. Comput. System Sci., 22 (1981), pp. 312–327.

[58] W. Paul, J. Seiferas, and J. Simon, *An information-theoretic approach to time bounds for on-line computation*, J. Comput. System Sci., 23 (1981), pp. 108–126.

[59] N. Pippenger, *On simultaneous resource bounds*, in Proc. 20th Foundations of Computer, 1979, pp. 307–311.

[60] N. Pippenger and M. Fischer, *Relations among complexity measures*, J. Assoc. Comput. Mach., 26 (1979), pp. 361–381.

[61] V. PRATT AND L. STOCKMEYER, *A characterization of the power of vector machines*, J. Comput. System Sci., 12 (1976), pp. 198–221.

[62] K. REGAN, *A new parallel vector model, with exact characterizations of* NC^k, in Proc. 11th Symposium on Theoretical Aspects of Computer Science, Lecture Notes in Comput. Sci., 778 (1994), pp. 289–300.

[63] R. REISCHUK, *A fast implementation of multidimensional storage into a tree storage*, Theoret. Comput. Sci., 19 (1982), pp. 253–266.

[64] W. RUZZO, *On uniform circuit complexity*, J. Comput. System Sci., 22 (1981), pp. 365–383.

[65] C. SCHNORR, *Satisfiability is quasilinear complete in NQL*, J. Assoc. Comput. Mach., 25 (1978), pp. 136–145.

[66] A. SCHÖNHAGE, *Storage modification machines*, SIAM J. Comput., 9 (1980), pp. 490–508.

[67] ———, *A nonlinear lower bound for random-access machines under logarithmic cost*, J. Assoc. Comput. Mach., 35 (1988), pp. 748–754.

[68] P. VAN EMDE BOAS, *Machine models and simulations*, in Handbook of Theoretical Computer Science, vol. A, J. V. Leeuwen, Ed., Elsevier, New York, MIT Press, Cambridge, MA, 1990, pp. 1–66.

[69] D. WILLARD, *A density control algorithm for doing insertions and deletions in a sequentially ordered file in a good worst-case time*, Inform. and Comput., 97 (1992), pp. 150–204.

SIAM J. COMPUT.
Vol. 25, No. 1, pp. 169–192, February 1996

ON THE COMPOSITION OF ZERO-KNOWLEDGE PROOF SYSTEMS*

ODED GOLDREICH† AND HUGO KRAWCZYK‡

Abstract. The wide applicability of zero-knowledge interactive proofs comes from the possibility of using these proofs as subroutines in cryptographic protocols. A basic question concerning this use is whether the (sequential and/or parallel) composition of zero-knowledge protocols is zero-knowledge too. We demonstrate the limitations of the composition of zero-knowledge protocols by proving that the original definition of zero-knowledge is not closed under sequential composition; and that even the strong formulations of zero-knowledge (e.g., black-box simulation) are not closed under parallel execution.

We present lower bounds on the round complexity of zero-knowledge proofs, with significant implications for the parallelization of zero-knowledge protocols. We prove that three-round interactive proofs and constant-round Arthur–Merlin proofs that are black-box simulation zero-knowledge exist only for languages in BPP. In particular, it follows that the "parallel versions" of the first interactive proofs systems presented for quadratic residuosity, graph isomorphism, and any language in NP, are not black-box simulation zero-knowledge, unless the corresponding languages are in BPP. Whether these parallel versions constitute zero-knowledge proofs was an intriguing open questions arising from the early works on zero-knowledge. Other consequences are a proof of optimality for the round complexity of various known zero-knowledge protocols and the necessity of using secret coins in the design of "parallelizable" constant-round zero-knowledge proofs.

Key words. zero-knowledge, cryptographic protocols, interactive proofs

AMS subject classifications. 68Q99, 94A60

1. Introduction.

In this paper we investigate the problem of composing zero-knowledge proof systems. Zero-knowledge proof systems, introduced in the seminal paper of Goldwasser, Micali, and Rackoff [GMR1], are efficient interactive proofs which have the remarkable property of yielding nothing but the validity of the assertion. Namely, whatever can be efficiently computed after interacting with a zero-knowledge prover, can be efficiently computed on input of a valid assertion. Thus, a zero-knowledge proof is computationally equivalent to an answer of a trusted oracle.

A basic question regarding zero-knowledge interactive proofs is whether their composition remains zero-knowledge. This question is not only of theoretical importance, but is also crucial to the utilization of zero-knowledge proof systems as subprotocols inside cryptographic protocols. Of particular interest are sequential and parallel composition. Candidate "theorems" (whose correctness we investigate) are listed here.

Sequential Composition. Let Π_1 and Π_2 be zero-knowledge proof systems for languages L_1 and L_2, respectively. Then, on input $x_1 \circ x_2$, first executing Π_1 on x_1 and afterwards executing Π_2 on x_2 constitutes a zero-knowledge interactive proof system for $L_1 \circ L_2$.

Parallel Composition. Let Π_1 and Π_2 be as above. Then, on input $x_1 \circ x_2$, concurrently executing Π_1 on input x_1 and Π_2 on x_2 constitutes a zero-knowledge interactive proof system for $L_1 \circ L_2$. (Concurrent execution means that the ith message of the composed protocol consists of the concatenation of the ith messages in Π_1 and Π_2, respectively.)

Sequential composition. Soon after the publication of [GMR1], several researchers noticed that the formulation of zero-knowledge proposed therein (hereafter referred as the *original* formulation) is probably not closed under sequential composition. In particular, Feige

*Received by the editors August 30, 1993; accepted for publication (in revised form) July 28, 1994. This research was partially supported by the Fund for Basic Research Administered by the Israeli Academy of Sciences and Humanities. A preliminary version of this paper appeared in the *Proc. 17th ICALP*, Lecture Notes in Computer Science, Vol. 443, Springer-Verlag, Berlin, 1990, pp. 268–282.

†Department of Computer Science, Technion, Haifa, Israel.

‡IBM T. J. Watson Research Center, Yorktown Heights, NY 10598 (hugo@watson.ibm.com). This research work was performed while this author was with the Department of Computer Science, Technion, Haifa, Israel.

and Shamir [Fei] proposed a protocol conjectured to be a counterexample to the Sequential Composition "Theorem." In this paper we use the ideas of [Fei] and new results on pseudorandom distributions [GK] to prove that, indeed, the *original formulation of zero-knowledge is not closed under sequential composition.* Our proof is independent of any intractability assumption. It applies to the notion of *computational* zero-knowledge (see §2), and uses computationally unbounded provers. (So far no proof exists for the same result with provers limited to polynomial time, or for statistical or perfect zero-knowledge.)

The reader should be aware that the Sequential Composition Theorem was proven (by Goldreich and Oren [GO], [Ore]) for a stronger ("nonuniform") formulation of zero-knowledge suggested by several authors (cf. [Fei], [GMR2], [GO], [Ore], and [TW]). The Sequential Composition Theorem is crucial to the utilization of zero-knowledge interactive proofs in cryptographic applications and in particular to the construction of cryptographic protocols for playing any computable game [Yao], [GMW2].

Parallel composition. Parallel composition of interactive proofs is widely used as a means of decreasing the error probability of proof systems, while maintaining the number of rounds. Of course, one would be interested in applying these advantages of parallelism to zero-knowledge protocols as well. Parallelism is also used in multiparty protocols in which parties wish to prove (the same and/or different) statements to various parties concurrently. Unfortunately, we show in this paper a counterexample to the Parallel Composition Theorem. Namely, we introduce a pair of protocols which are (computational) zero-knowledge with respect to the strongest known definitions (including the nonuniform formulation discussed above and the "black-box simulation" formulation discussed below) yet their parallel composition is not zero-knowledge (not even in the "weak" sense of the original [GMR1] formulation). Also in this case, our proof does not rely on any unproven hypotheses; on the other hand, it uses in an essential way the unbounded computational power of the prover and the computational notion of zero-knowledge. Based on intractability assumptions, Feige and Shamir [FS2] show a perfect zero-knowledge protocol with a polynomial-time prover which fails parallel composition. Our results below on three-round zero-knowledge proofs imply a similar result, but our case requires a superlogarithmic number of repetitions, while in [FS2] two repetitions suffice.

By the above result we have ruled out the possibility of proving that particular interactive proofs are zero-knowledge by merely arguing that they are the result of parallel composition of various zero-knowledge protocols. But this does not resolve the question of whether concrete cases of composed interactive proofs are zero-knowledge. In particular, since the first presentation of the results in [GMR1] and [GMW1], it was repeatedly asked whether the "parallel versions" of the interactive proofs presented for quadratic residuosity, graph isomorphism, and any language in NP are zero-knowledge.

Round complexity of zero-knowledge proofs. In this paper we prove a general result concerning the round complexity of zero-knowledge interactive proofs which, in particular, resolves the question of parallelization of the above-mentioned protocols. This general result states that *only BPP languages have three-round interactive proofs which are black-box simulation zero-knowledge.*[1] Since the parallel versions of the above examples are three-round interactive proofs (with negligible cheating probability for the prover) it follows that these interactive proofs cannot be proven zero-knowledge using black-box simulation zero-knowledge, unless the corresponding languages are in BPP. This (negative) result is proven for computa-

[1] This result applies to interactive proofs in which the prover can convince the verifier of accepting a false assertion with only negligible probability. The above-mentioned languages have three-round zero-knowledge interactive proofs in which the prover has a significant (e.g., constant) probability of cheating.

tional zero-knowledge proofs and therefore applies to statistical and perfect zero-knowledge as well.

Loosely speaking, we say that an interactive proof for a language L *is black-box simulation zero-knowledge* if there exists a (probabilistic polynomial-time) universal simulator which, using any (even nonuniform) verifier V^* as a black box, produces a probability distribution which is polynomially indistinguishable from the distribution of conversations of (the same) V^* with the prover, on inputs in L. This definition of zero-knowledge is more restrictive than the original one, which allows each verifier V^* to have a specific simulator S_{V^*}. Nevertheless, all *known* zero-knowledge protocols are also black-box simulation zero-knowledge. This fact cannot come as a surprise since it is hard to conceive of a way of taking advantage of the full power of the more liberal definition.

It is not plausible that our result could be extended to four-round interactive proofs since such proofs are known for languages believed to be outside BPP. The protocols for quadratic nonresiduosity [GMR1] and graph nonisomorphism [GMW1] are such examples. In addition, zero-knowledge interactive proofs of five rounds are known for quadratic residuosity and graph isomorphism [BMO1], and, assuming the existence of claw-free permutations, there exist five-round zero-knowledge interactive proofs for any language in NP [GKa]. Moreover, our results extend to zero-knowledge *arguments*[2], for which Feige and Shamir [FS1] presented (assuming the existence of one-way functions) a four-round protocol for any language in NP. Our result implies that the round complexity of this protocol is optimal (as long as BPP $\neq$ NP).

Constant-round Arthur–Merlin proofs. When restricting ourselves to Arthur–Merlin interactive proofs, we can extend the above result to any constant number of rounds. We show that *only BPP languages have constant-round Arthur–Merlin proofs which are also black-box simulation zero-knowledge.*

Arthur–Merlin interactive proofs, introduced by Babai [Bab], are interactive proofs in which all the messages sent by the verifier are the outcome of his coin tosses. In other words, the verifier "keeps no secrets from the prover." Our result is a good reason to believe that the only feasible way of constructing constant-round zero-knowledge interactive proofs is to let the verifier use "secret coins." Indeed, the above-mentioned constant-round zero-knowledge proofs, as well as the constant-round statistical zero-knowledge proofs of [BMO2], use secret coins. Thus, secret coins do help in the zero-knowledge setting. This should be contrasted with the result of Goldwasser and Sipser [GS], which states that Arthur–Merlin interactive proofs are *equivalent* to general interactive proofs (as far as language recognition is concerned). They show that any language having a general interactive proof (IP) of k rounds also has an Arthur–Merlin (or AM) proof of k rounds. Using our result we see that in the zero-knowledge setting such a preservation of rounds (when transforming IP into AM) is not plausible (e.g., graph nonisomophism).

Our result concerning Arthur–Merlin proofs is tight in the sense that the languages considered above (e.g., graph nonisomorphism, every language in NP) have unbounded (i.e., $\omega(n)$-round, for every unbounded function $\omega : N \to N$) Arthur–Merlin proof systems which are black-box simulation zero-knowledge. In particular, we get that bounded-round Arthur–Merlin proofs which are black-box zero-knowledge exist only for BPP, while unbounded round proofs of the same type exist for all PSPACE (if one-way functions exist [IY], [B*], [Sha]). That is, while the *finite hierarchy* of languages having black-box zero-knowledge Arthur–

[2] Interactive *arguments* (also known as "computationally sound proofs" and "computationally convincing protocols") differ from an interactive proof system in that the soundness condition of the system is formulated with respect to *probabilistic polynomial-time* provers, possibly with auxiliary input (see [BCC]). Namely, *efficient* provers cannot fool the verifier into accepting an input not in the language, except with negligible probability.

Merlin proofs collapses to BPP (= AM(0)), the corresponding *infinite hierarchy* contains all of PSPACE. This implies (assuming the existence of one-way functions) a separation between the two hierarchies.

Organization. In §2 we outline the definitions of interactive proofs and zero-knowledge, and introduce some terminology and notation used throughout the paper. Section 3 presents the definitions and results concerning pseudorandom distributions that are used for disproving the composition theorems. In §§4 and 5 we present these disproofs for the case of sequential and parallel composition, respectively. Finally, in §6 we present the lower bounds on the round complexity of black-box simulation zero-knowledge proofs. We stress that this last section is technically independent from §§3, 4, and 5 and can be read without going through these sections.

2. Preliminaries. The notions of interactive proofs and zero-knowledge were introduced by Goldwasser, Micali, and Rackoff [GMR1]. Here, we give an informal outline of these notions. For formal and complete definitions, as well as the basic results concerning these concepts, the reader is referred to [GMR1], [GMW1], and [GMR2].

An *interactive proof* is a two-party protocol in which a computationally unrestricted *prover*, P, interacts with a probabilistic polynomial-time *verifier*, V, by exchanging messages. Both parties share a common input x. At the end of the interaction the verifier computes a predicate depending on this input and the exchanged messages in order to *accept* or *reject* the input x. Such a protocol, denoted $\langle P, V\rangle$, is called an *interactive proof for a language L* if the following two conditions hold.

Completeness property. For any positive constant c and sufficiently long $x \in L$, Prob(V accepts x) $> 1 - |x|^{-c}$.

Soundness property. For any positive constant c and sufficiently long $x \notin L$, Prob(V accepts x) $< |x|^{-c}$, no matter how the prover behaves during the protocol.

(The above probabilities are taken over the coin tosses of both the prover and the verifier.) In other words, we require that on inputs belonging to L the probability that the prover P "convinces" V to accept the common input is almost 1, while on inputs outside L there is no prover that can fool V into accepting, except with negligible probability.

Note. Notice that we define an interactive proof to have a negligible probability of error. Some authors define this probability to be just a constant (e.g., $\frac{1}{3}$). The latter is motivated by the fact that constant-error interactive proofs can be converted into negligible-error proofs by parallel repetition. However, in the setting of zero-knowledge interactive proofs, our results show that such parallel repetition may sacrifice the zero-knowledge property.

An interactive proof in which the *honest* verifier chooses all its messages at random (i.e., with uniform probability over the set of all strings of the same length as the message) is called an *Arthur–Merlin* interactive proof [Bab]. That is, in an Arthur–Merlin proof system the only nontrivial computation carried out by the honest verifier is the evaluation of a polynomial-time predicate at the end of the interaction, in order to decide the acceptance or rejection of the input to the protocol. We say that such a verifier uses *public coins.* (Notice that there is no "public coin" restriction on the cheating verifiers.)

We say that an interactive proof has k rounds if there are a total of k messages (alternately) exchanged between the prover and verifier during the protocol (i.e., we count messages from both parties). In general, the number k can be a function $k(|x|)$ of the input length. The notation IP(k) stands for the class of languages having k-round interactive proofs, and AM(k) stands for languages having k-round Arthur–Merlin interactive proofs.

An interactive proof is called *zero-knowledge* if on input $x \in L$ no probabilistic polynomial-time verifier (i.e., one that may arbitrarily deviate from the predetermined program) gains information from the execution of the protocol except the knowledge that x belongs to L.

This means that any polynomial-time computation based on the conversation with the prover can be simulated, without interacting with the real prover, by a probabilistic polynomial-time machine ("the simulator") that gets x as its only input. More precisely, let $\langle P, V^*\rangle(x)$ denote the probability distribution generated by the interactive machine (verifier) V^*, which interacts with the prover P on input $x \in L$. We say that an interactive proof is *zero-knowledge* if for all probabilistic polynomial-time machines V^*, there exists a probabilistic expected polynomial-time algorithm M_{V^*} (called the *simulator*) that on inputs $x \in L$ produces probability distributions $M_{V^*}(x)$ that are polynomially indistinguishable from the distributions $\langle P, V^*\rangle(x)$. (This notion of zero-knowledge is also called *computational zero-knowledge*.)[3]

The above formalization of the notion of zero-knowledge is taken from the original paper by Goldwasser, Micali, and Rackoff [GMR1]. Later, stronger formulations of zero-knowledge were introduced in which the simulation requirement is extended to deal with stronger verifiers [Fei], [GMR2], [GO], [Ore], [TW], namely, verifiers with nonuniform properties (e.g., probabilistic polynomial-time verifiers that get an additional *auxiliary-input* tape), or verifiers modeled by polynomial-size circuits.

One further formulation of zero-knowledge is called *black-box simulation zero-knowledge* [GO], [Ore]. This formulation differs from the former by requiring the existence of a ("universal") simulator that, using any (nonuniform) verifier V^* as a black box, succeeds in simulating the $\langle P, V^*\rangle$ interaction. In other words, there exists a probabilistic expected polynomial-time oracle machine M such that for any polynomial-size verifier V^* and for $x \in L$, the distributions $\langle P, V^*\rangle(x)$ and $M^{V^*}(x)$ are polynomially indistinguishable.

Remark. A complete formalization of the notion of black-box simulation zero-knowledge requires dealing with the following technical problem. The simulator uses V^* as a black box. This means that the simulator is responsible, during the simulation process, for feeding into the black box the external parameters that determine the behavior of V^*. These parameters are the string representing the input to the protocol, the contents of the random tape of V^*, and the messages of the prover. A problem arises when feeding the random coins used by V^*. Although the number of coin tosses used by a particular verifier V^* is bounded by a polynomial, there is no *single* polynomial that bounds this number for *all* possible verifiers. On the other hand, the simulator M runs (expected) time that is bounded by a specific polynomial. So, how can this simulator manage to feed a verifier requiring more coin tosses than this bound? In [BMO2] this problem is overcome by stating the existence of two random tapes for M. The first is used in the regular way for M's computations. The second can be entirely fed by M into V^* in a single step. That is, M can feed the random coins for the black box in an "intelligent way" as long as the number of coins does not exceed the time capability of M; beyond this number it can only feed truly random bits. We stress that this formalization is general enough to include all *known* zero-knowledge proofs.

An alternative solution to the above problem is to have, *for each* polynomial p, a simulator M_p which simulates all verifiers V^* that use at most $p(|x|)$ random coins on any input x. Clearly, the running time of the simulator M_p may depend on the polynomial p, and then the above difficulty is overcome. This second formulation is weaker than the one proposed in [BMO2], but it suffices for the results proved in our paper and is therefore adopted here. (In fact, our results in §6 only require the existence of a simulator that simulates deterministic verifiers, i.e., M_p with $p \equiv 0$.)

Based on the above remark we give our definition of black-box simulation zero-knowledge.

[3] Other definitions were proposed in which it is required that the distribution generated by the simulator be *identical* to the distribution of conversations between the verifier and the prover (*perfect* zero-knowledge), or at least statistically close (*statistical* zero-knowledge). See [GMR2] for further details.

DEFINITION. *An interactive proof $\langle P, V \rangle$ is called* black-box simulation zero-knowledge *if for every polynomial p, there exists a probabilistic expected polynomial-time oracle machine M_p such that for any polynomial-size verifier V^* that uses at most $p(n)$ random coins on inputs of length n, and for $x \in L$, the distributions $\langle P, V^* \rangle(x)$ and $M_p^{V^*}(x)$ are polynomially indistinguishable.*

Note. The notion of polynomial indistinguishability in the above definition can be formalized based on nonuniform polynomial-size distinguishers or uniform polynomial-time distinguishers which have black-box access to the corresponding V^*. Our results apply to both formalizations.

Terminology. Throughout this paper we use the term *negligible* to denote functions that are (asymptotically) smaller than 1 over any polynomial, and the term *nonnegligible* to denote functions that are greater than 1 over some fixed polynomial. More precisely, a function f from nonnegative integers to reals is called *negligible* if for all constants c and sufficiently large n, $f(n) < n^{-c}$. The function f is called *nonnegligible* if there exists a constant c such that for all (sufficiently large) n, $f(n) > n^{-c}$. (Observe that nonnegligible is not the complement of negligible.)

Notation. We use the notation $e \in_R S$ for "the element e is chosen with uniform probability from the set S."

3. On evasive and pseudorandom sets. In the demonstration of counterexamples for the "composition theorems" we make use of pseudorandom distributions which have some interesting "evasiveness" properties. These properties and the corresponding proofs are given in [GK] and cited here without proof.

Roughly speaking, a distribution on a set of strings of length k is *pseudorandom* if this distribution cannot be efficiently (i.e., in time polynomial in k) distinguished from the uniform distribution on the set of all strings of length k. In order to formalize this notion one has to define it in asymptotical terms and refer to collections of distributions (called *pseudorandom ensembles*), rather than single distributions. The notion of a "pseudorandom set" is made precise in the following definition.

DEFINITION 3.1. *A set $S \subseteq \{0,1\}^k$ is called $(\tau(k), \varepsilon(k))$-*pseudorandom *if for any (probabilistic) circuit C of size $\tau(k)$ with k inputs and a single output*

$$|p_C(S) - p_C(\{0,1\}^k)| \le \varepsilon(k),$$

where $p_C(S)$ (resp., $p_C(\{0,1\}^k)$) denotes the probability that C outputs 1 when given elements of S (resp., $\{0,1\}^k$), chosen with uniform probability.

Note that a collection of uniform distributions on a sequence of sets $S_1, S_2, \ldots,$ where each S_k is a $(\tau(k), \varepsilon(k))$-pseudorandom set, constitutes a pseudorandom ensemble, provided that both functions $\tau(n)$ and $\varepsilon^{-1}(n)$ grow faster than any polynomial. Therefore, we shall refer to such a sequence of sets as a *pseudorandom ensemble.*

We now present the concept of "evasive sets." Informally, evasiveness means that it is hard, for efficient algorithms, to find strings which belong to these sets.

DEFINITION 3.2. *Let $S_1, S_2, \ldots$ be a sequence of (nonempty) sets such that for every n, $S_n \subseteq \{0,1\}^{Q(n)}$, for a fixed polynomial Q. Such a sequence is called a* polynomially evasive *(denoted* P-evasive*) ensemble if for any probabilistic polynomial-time algorithm A, any constant c, any sufficiently large n, and any $x \in \{O, 1\}^n$,*

$$\text{Prob}(A(x) \in S_n) < n^{-c},$$

where the probability is taken over the random coins of algorithm A.

The following theorem states the existence of a P-evasive ensemble which is also pseudorandom.

THEOREM 3.1 [GK]. *There exists a* P*-evasive pseudorandom ensemble* $S_1, S_2, \ldots$ *with* $Q(n) = 4n$*. Furthermore, there exists a Turing machine which on input* 1^n *outputs the set* S_n*.*

For disproving the parallel composition theorem we shall need a stronger notion of evasiveness. Namely, one which also resists nonuniform algorithms. This definition of evasiveness involves a collection of sets for each length, rather than a single set per length as in the uniform case.

DEFINITION 3.3. *Let* $Q(\cdot)$ *be a polynomial, and for* $n = 1, 2, \ldots$ *let* $\mathcal{S}^{(n)}$ *be a collection of* 2^n *sets* $\{S_1^{(n)}, \ldots, S_{2^n}^{(n)}\}$*, where each* $S_i^{(n)} \subseteq \{0, 1\}^{Q(n)}$*. The sequence* $\mathcal{S}^{(1)}, \mathcal{S}^{(2)}, \ldots$ *is called a* nonuniform polynomially evasive (*denoted* P/poly-evasive) *ensemble if for any* $c > 0$*, sufficiently large* n*, and any (probabilistic) circuit* C *of size* n^c *(with* n *inputs and* $Q(n)$ *outputs)*

$$\text{Prob}(C(i) \in S_i) < \frac{1}{n^c},$$

where the probability is taken over the random coins of C *and* $i \in \{1, \ldots, 2^n\}$*, both with uniform probability.*

That is, a sequence $\mathcal{S}^{(1)}, \mathcal{S}^{(2)}, \ldots$ is a P/poly-evasive ensemble if any circuit of size polynomial in n, which gets a randomly selected index of one of the sets in $\mathcal{S}^{(n)}$, cannot succeed in outputting an element in that set, except for a negligible probability.

Remark. Notice that in the definition of P-evasive ensembles the (uniform) algorithm trying to hit an element in the evasive set S_n gets as input a string x of length n, which can be seen as an auxiliary input. The crucial difference between this "uniform" definition and the definition of P/poly-evasiveness is that in the latter the auxiliary input is allowed to be of any length polynomial in the length of the target strings, while in the former the auxiliary input is properly shorter than the target strings in the set S_n.

The following theorem states the existence of a P/poly-evasive ensemble which is composed of pseudorandom sets.

THEOREM 3.2. *There exists a* P*/poly-evasive ensemble* $\mathcal{S}^{(1)}, \mathcal{S}^{(2)}, \ldots$ *with* $Q(n) = 4n$*, such that for every* n*, each* $S_i^{(n)}$ *is a* $(2^{n/4}, 2^{-n/4})$*-pseudorandom set of cardinality* 2^n*. Furthermore, there exists a Turing machine which on input* 1^n *outputs the collection* $\mathcal{S}^{(n)}$*.*

The proof of this theorem is given in the appendix.

4. Sequential composition of zero-knowledge protocols.

A natural requirement from the notion of zero-knowledge proofs is that the information obtained by the verifier during the execution of a zero-knowledge protocol will not enable him to extract any additional knowledge from subsequent executions of the same protocol. That is, it would be desirable for the *sequential composition* of zero-knowledge protocols to yield a protocol which is itself zero-knowledge. Such a property is crucial for applications of zero-knowledge protocols in cryptography (for details and further motivation, see [GO] and [Ore]).

We prove that the original definition of (computational) zero-knowledge introduced by Goldwasser, Micali, and Rackoff in [GMR1], *is not closed* under sequential composition. Several authors have previously observed that this definition *probably* does not guarantee the robustness of zero-knowledge under sequential composition, and hence have introduced more robust formulations of zero-knowledge [Fei], [GMR2], [GO], [Ore], [TW]. But so far, no proof has been given for the claim that computational zero-knowledge (with uniform verifiers) fails under sequential composition.

Intuitively, the reason that a zero-knowledge protocol could not be closed under sequential composition is that the definition of zero-knowledge requires that the information transmitted in the execution of the protocol is "useless" for any *polynomial-time computation*; it does not rule out the possibility that a cheating verifier could take advantage of this "knowledge" in a subsequent interaction with the (*nonpolynomial time*) prover for obtaining valuable information. This intuition (presented in [Fei]) is the basis of our example of a protocol which is zero-knowledge in a single execution but reveals significant information when composed twice in a sequence. This protocol, presented in the proof of the following theorem, uses a P-evasive ensemble as defined in Definition 3.2 and whose existence is stated in Theorem 3.1.

THEOREM 4.1. *Computational zero-knowledge* ([GMR1]*formulation*) *is not closed under sequential composition.*

Proof. Let $S_1, S_2, \ldots$ be a P-evasive pseudorandom ensemble as described in Theorem 3.1. Also, let K be an (arbitrary) hard Boolean function, in the sense that the language $L_K = \{x : K(x) = 1\}$ is not in BPP (we use this function as a "knowledge" function).

We present the following interactive-proof protocol $\langle P, V \rangle$ for the language $L = \{0, 1\}^*$. (Obviously, this language has a trivial zero-knowledge proof in which the verifier accepts every input without carrying out any interaction. We intentionally modify this trivial protocol in order to demonstrate a zero-knowledge protocol which fails sequential composition.)

Let x be the common input for P and V, and let n denote the length of x. The verifier V begins by sending to the prover a random string s of length $4n$. The prover P checks whether $s \in S_n$ (the nth set in the P-evasive ensemble defined above). If this is the case (i.e., $s \in S_n$), then P sends to V the value of $K(x)$. Otherwise (i.e., $s \notin S_n$), P sends to V a string s_0 randomly selected from S_n. In any case the verifier accepts the input x (as belonging to L).

We stress that the same P-evasive ensemble is used in all the executions of the protocol. Thus, the set S_n does not depend on the specific input to the protocol, but only on its length. Therefore, the string s_0, obtained by the verifier in the first execution of the protocol, enables him to deviate from the protocol during a second execution in order to obtain the value of $K(x')$, for any x' of length n (and in particular for $x' = x$). Indeed, consider a second execution of the protocol, this time on input x'. A "cheating" verifier, which sends the string $s = s_0$ instead of choosing it at random, will get the value of $K(x')$ from the prover. Observe that this cheating verifier obtains information that it could not compute by itself. There is no way to simulate in probabilistic polynomial time the interaction in which the prover sends the value of $K(x')$; otherwise the language L_K would be in BPP (indeed, such a simulator could be used as a probabilistic polynomial-time algorithm for computing the function K with negligible error probability. To see that, notice that the real prover in an interaction with the above cheater verifier on inputs (x, x') will output $k(x')$ with probability 1. Therefore, the simulator must output the correct value of $k(x')$ with probability almost 1, or otherwise, its output is polynomially distinguishable from the real conversations). Thus, the protocol is not zero-knowledge when composed twice.

On the other hand, the protocol is zero-knowledge (when executed once). To show this, we present for any verifier V^*, a polynomial-time simulator M_{V^*} that can simulate the conversations between V^* and the prover P. There is only one message sent by the prover during the protocol. It sends the value of $K(x)$ when the string s sent by the verifier belongs to the set S_n, and a randomly selected element of S_n otherwise. By the evasivity condition of the set S_n, there is only a negligible probability that the first case holds. Indeed, no probabilistic polynomial-time machine (in our case, the verifier) can find such a string $s \in S_n$, except with negligible probability (no matter what the input x to the protocol is). Thus, the simulator can succeed by always simulating the second possibility, i.e., the sending of a random element s_0 from S_n. This step is simulated by randomly choosing s_0 from $\{0, 1\}^{4n}$ rather than from S_n. The indistinguishability of this choice from the original one follows from the fact that each S_n is a pseudorandom subset of $\{0, 1\}^{4n}$, and that the prover chooses s_0 from S_n with uniform probability. □

Remark. The argument presented in the above proof generalizes to any language L having a zero-knowledge interactive proof. Simply modify the zero-knowledge proof for L as in the proof of Theorem 4.1.

Remark. Another example of a zero-knowledge protocol which is not closed under sequential composition was independently found by D. Simon [Sim]. His construction assumes the existence of secure encryption systems.

5. Parallel composition of zero-knowledge protocols. In this section we address the question of whether zero-knowledge interactive proofs are robust under parallel composition.

Clearly, we cannot expect the original Goldwasser–Micali–Rackoff (GMR) definition to satisfy this condition: it is easy to see that a zero-knowledge protocol which is not closed under sequential composition can be transformed into another zero-knowledge protocol which fails parallel composition.

In light of the fact that *auxiliary-input* zero-knowledge is robust under sequential composition [GO], [Ore], it is an interesting open question whether this formulation of zero-knowledge is also robust under parallel composition. The main result of this section is that this is *not* the case. We prove the existence of protocols which are zero-knowledge even against nonuniform verifiers (e.g., auxiliary-input zero-knowledge), but which do not remain zero-knowledge when executed twice in parallel. As in the case of sequential composition our results concern only computational zero-knowledge.

The ideas used for the design of a protocol which fails parallel composition are similar to those used for the sequential case. There, we have used the pseudorandomness and evasiveness of some sets to construct the intended protocol. We also use this method here. The main difficulty of extending these properties to the present case is that now we need an evasive collection which resists even nonuniform verifiers. Clearly, a P-evasive ensemble will not satisfy this condition, since for any set of strings there exist nonuniform verifiers which can output elements in this set (e.g., by getting such a string as auxiliary input). Instead, we use the notion of P/poly-evasive ensembles as defined in Definition 3.3. Based on Theorem 3.2, which states the existence of such ensembles, we prove the main result of this section.

THEOREM 5.1. *Computational zero-knowledge* (*even with nonuniform verifiers*) *is not closed under parallel composition.*

Proof. We present a pair of protocols $\langle P_1, V_1\rangle$ and $\langle P_2, V_2\rangle$ which are zero-knowledge when executed independently, but whose parallel composition is provably not zero-knowledge.

We use some dummy steps in the protocols in order to achieve synchronization between them. Of course, one can modify the protocol, substituting these extra steps by significant ones. The version we give here prefers simplicity over naturality. Both protocols consist of five steps and are described below (see also Fig. 1).

The first protocol is denoted $\langle P_1, V_2\rangle$. Let x be the input to the protocol and let n denote its length. The protocol uses (for all its executions) a P/poly-evasive ensemble $S^{(1)}, S^{(2)}, \ldots$ with the properties described in Theorem 3.2. It also involves a hard Boolean function K as in the proof of Theorem 4.1. The prover P_1 begins by sending to V_1 an index $i \in_R \{1, \ldots, 2^n\}$. After two dummy steps the verifier V_1 sends to P_1 a string $s \in_R \{0, 1\}^{4n}$. The prover P_1 checks whether $s \in S_i^{(n)}$. If this is the case then it sends to V_1 the value of $K(x)$, (otherwise an empty message). This concludes the protocol.

The second protocol $\langle P_2, V_2\rangle$ uses the *same* P/poly-evasive ensemble $S^{(1)}, S^{(2)}, \ldots$ as protocol $\langle P_1, V_1\rangle$ does. The first step of the protocol is a dummy one. In the second step the verifier V_2 sends to P_2 an index $j \in_R \{1, \ldots, 2^n\}$. The prover P_2 responds with a string $r \in_R S_j^{(n)}$. After two more dummy steps the protocol stops.

We show that each of the above protocols is indeed zero-knowledge (even for nonuniform verifiers). For the first protocol, there are two steps of the prover to be simulated. In the

P_1	V_1	step	P_2	V_2
$i \in_R \{1, \cdots, 2^n\} \rightarrow$		1	dummy step	
	dummy step	2		$\leftarrow j \in_R \{1, \cdots, 2^n\}$
dummy step		3	$r \in_R S_j^{(n)} \rightarrow$	
	$\leftarrow s \in_R \{0,1\}^{4n}$	4		dummy step
if $s \in S_i^{(n)}$: $K(x) \rightarrow$		5	dummy step	

FIG. 1. *Protocols $\langle P_1, V_1\rangle$ and $\langle P_2, V_2\rangle$ with input x.*

first step P_1 sends an index $i \in_R \{1, \dots, 2^n\}$. The simulator does the same. In the second step, the prover sends the value of $K(x)$ *only* if the verifier succeeds in presenting him with a string which belongs to the set $S_i^{(n)}$. By the evasivity condition of the sequence $S^{(1)}, S^{(2)}, \dots$, this will happen with negligible probability and therefore the simulator can always simulate this step as for the case where the verifier sends a string $s \notin S_i^{(n)}$. (Observe that the circuits in the definition of P/poly-evasive ensembles only get as input the index of the set to be hit. Nevertheless, in our case the circuits also have an additional input x. Clearly, this cannot help them find an element in $S_i^{(n)}$; otherwise, circuits which have such a string incorporated will contradict the evasiveness condition.)

In the second protocol, $\langle P_2, V_2\rangle$, the only significant step of the prover P_2 is when it sends an element $r \in_R S_j^{(n)}$ in response to the index j sent by the verifier. In this case the simulator will send a string $r' \in_R \{0, 1\}^{4n}$. Using the pseudorandomness property of the set $S_j^{(n)}$ we get that the simulator's choice is polynomially indistinguishable from the prover's one.

Finally, we show that the parallel composition of the above protocols into a single protocol $\langle P, V\rangle$ is not zero-knowledge. Let V^* be a "cheating" verifier which behaves as follows. Instead of sending a randomly selected index j (corresponding to the second step of the subprotocol $\langle P_2, V_2\rangle$) it sends the index i received from P as part of P_1's first step. Thus, $j = i$, and the prover P will respond with a string $r \in S_i^{(n)}$. In the next step this V^* will send string r to P instead of the "random" string s that V_1 should send to P_1. The prover P will verify that $r \in S_i^{(n)}$ and then will send the information $K(x)$. By the hardness of the function K this step cannot be simulated by a probabilistic polynomial-time machine. Therefore, the composed protocol $\langle P, V\rangle$ is not zero-knowledge. □

Remark. The two protocols $\langle P_1, V_1\rangle$ and $\langle P_2, V_2\rangle$ can be merged into a single zero-knowledge protocol which is not robust under parallel composition. In this merged protocol, the verifier chooses (at random) an index $i \in \{1, 2\}$, sends it to the prover, and then both parties execute the protocol $\langle P_i, V_i\rangle$. When executing two copies of this protocol in parallel, the verifiers may choose $i = 1$ and $i = 2$, respectively, thus forcing a parallel execution of $\langle P_1, V_1\rangle$ and $\langle P_2, V_2\rangle$, which we have shown not to be zero-knowledge.

6. On the round complexity of zero-knowledge proofs. In this section we present lower bounds on the round complexity of black-box simulation zero-knowledge interactive proofs. We show that only languages in BPP have constant-round Arthur–Merlin interactive proofs which are *black-box simulation zero-knowledge.* (For a definition of black-box simulation zero-knowledge and Arthur–Merlin interactive proofs, see §2.) We have the following theorem.

THEOREM 6.1. *A language L has a constant-round Arthur–Merlin interactive proof which is black-box simulation zero-knowledge if and only if $L \in BPP$.*

In §6.1 we present a proof for a special case of this theorem, namely, for the case of a three-round Arthur–Merlin protocol. The general case is proved in §6.2 using careful extensions of the ideas presented for this special case.

The three-round case can also be extended to general interactive proof systems. That is, we also have the following theorem, proved in §6.3.

THEOREM 6.2. *A language L has a three-round interactive proof which is black-box simulation zero-knowledge if and only if $L \in BPP$.*

(We remark that [GO] and [Ore] show that two-round (auxiliary-input) zero-knowledge proofs—not necessarily black-box simulation—exist only for BPP languages.)

Our results are optimal in the sense that there exist Arthur–Merlin interactive proofs, for languages believed to be outside BPP, with unbounded number of rounds and which are black-box simulation zero-knowledge. Similarly, there exist four-round interactive proof protocols (using private coins) which are also black-box simulation zero-knowledge. For further details about these protocols, and some consequences concerning the hierarchy of languages having zero-knowledge Arthur–Merlin proofs, see §1.

It is interesting to note that our results hold also for a weaker notion of black-box simulation zero-knowledge, namely, one which only requires the existence of a black-box simulator that succeeds in simulating conversations with *deterministic* (nonuniform) verifiers. The sufficiency of this condition follows from the proofs below. Also, the formulation of the completeness condition of an interactive proof (see §2) can be relaxed in the following way. We have defined the completeness condition by requiring that the prover convince the verifier of accepting an input in the language with probability almost 1 (i.e., 1 minus a negligible fraction). For the correctness of our results it suffices to require just a nonnegligible probability. (In this section we use this weaker formulation of the completeness condition.) On the other hand, the requirement of a negligible probability of convincing the verifier to accept an input not in the language (the soundness condition) is essential. (For example, three-round zero-knowledge protocols exist for all languages in NP if the soundness condition is formulated with probability $\frac{1}{2}$ [GMW1].) Finally, our results hold also in the setting of *interactive arguments* [BCC], i.e., "interactive proofs" in which the prover is limited to probabilistic polynomial-time computations, possibly getting an auxiliary input.

6.1. The case AM(3).

The protocol $\langle \mathbf{P}, \mathbf{V} \rangle$. Consider an Arthur–Merlin protocol $\langle P, V \rangle$ for a language L, consisting of three rounds. We use the following notation. Denote by x the input for the protocol, and by n the length of this input. The first message in the interaction is sent by the prover. We denote it by α. The second round is the V, which sends a string β. The third (and last) message is from P, and we denote it by γ. The predicate computed by the verifier V in order to accept or reject the input x is denoted by ρ_V, and we consider it, for convenience, as a deterministic function $\rho_V(x, \alpha, \beta, \gamma)$. (For the general case, see Remark 6.2.) We will also assume, without loss of generality, the existence of a polynomial $l(n)$ such that $|\alpha| = |\beta| = l(n)$.

The simulation process. Let this three-round Arthur–Merlin protocol $\langle P, V \rangle$ be black-box simulation zero-knowledge. Denote by M the guaranteed probabilistic expected polynomial-time black-box simulator which, given access to the black-box V^*, simulates $\langle P, V^* \rangle$. The process of simulation consists of several "tries" or calls to the interacting verifier V^* ("the black box"). In each such call the simulator M feeds the arguments for V^*. These arguments are the input y (which may be different from the "true" input x), the random coins for V^*, and a string α representing the message sent by the prover P. In our case, it suffices for our results to consider a simulator that is just able to simulate conversations with *deterministic* (*nonuniform*) *verifiers*. In particular, this simulator does not care about feeding the black-box V^* with random coins. This simplifies our proof by avoiding any reference to these random coins for V^*, and strengthen our result (since it holds even under the sole existence of this weak kind of simulator).

After completing its tries the simulator outputs a conversation $(y, \alpha, \beta, \gamma)$.

We shall make some further simplifying assumptions on the behavior of the simulator M, which will not restrict the generality. In particular, we assume that some cases, which may arise with only negligible probability, do not happen at all. This cannot significantly effect the success probability of the simulator. In other words, any black-box simulator which successfully simulates $\langle P, V^* \rangle$ conversations of deterministic verifiers V^* can be changed into another simulator for which the following conventions hold and which has the same success probability as the original simulator, except for a possibly negligible difference. We assume the following:

- The conversations output by M always have the form $(x, \alpha, \beta, \gamma)$ (i.e., $y = x$), and that the string β equals the message output by V^* when fed with inputs x and α. Note that these conditions always hold for the real conversations generated by the prover P and the (deterministic) verifier V^*. Therefore, the simulator must almost always do the same. (Otherwise, a distinguisher which has access to V^* would distinguish between the simulator's output and the original conversations.)
- The simulator M explicitly tries, in one of its calls to V^*, the arguments x and α appearing in the output conversation. (For example, once the simulator decides on the output conversation with a specific parameter α, it explicitly feeds V^* with x and this value of α, regardless of whether it asked α before or not. In any case, the answer of the deterministic V^* to the pair (x, α), will be always the same.)
- The simulator runs in (strictly) polynomial time. (In Remark 6.1 below we show how to handle the general case in which the simulator runs in *expected* polynomial time.) We denote by $t(n)$ a polynomial bounding the number of calls tried by M before outputting a conversation.

The simulator as a subroutine. Our goal is to present a BPP algorithm for the language L. The idea is to use the simulator M in order to distinguish between inputs in and outside L. For that, we use the simulator itself as a subroutine of the BPP algorithm. We do not make any assumption on the internal behavior of this simulator, but just use the following observation. *The behavior of the simulator M, interacting with a verifier V^*, is completely determined by the input x, the random tape R_M used by M, and the strings output by V^** (*in response to the arguments fed by the simulator during its tries*). Therefore, in order to operate M, we just need to feed it with an input x, a tape of random coins, and a sequence of responses to its messages α. Below we formally describe a computation process that uses M as a subroutine. (We stress that in this process there is no *explicit* verifier present.)

Fix an input x of length n, a string R_M (of length $q(n)$, where $q(\cdot)$ is a polynomial bounding the number of random coins used by M on inputs of length n), and $t = t(n)$ (arbitrary) strings $\beta^{(1)}, \beta^{(2)}, \ldots, \beta^{(t)}$, each of length $l(n)$. Activate M on input x with its random tape containing R_M. For each y and α tried by M, respond with a message β from the above list $\beta^{(1)}, \beta^{(2)}, \ldots, \beta^{(t)}$ according to the following rule. (This rule depends on the strings α but not on y.) To the first α presented by M respond with $\beta^{(1)}$. For subsequent α's check whether the same string α was presented by M. If so, respond with the same β as in that case; if it is the first time this α is presented then respond with the first unused $\beta^{(i)}$ in the list. That is, if α is the ith *different* string presented by M then we respond with $\beta^{(i)}$. We denote the ith different α by $\alpha^{(i)}$. Clearly, $\alpha^{(i)}$ is uniquely determined by x, R_M, and the $i-1$ strings $\beta^{(1)}, \ldots, \beta^{(i-1)}$, i.e., there exists a deterministic function α_M such that $\alpha^{(i)} = \alpha_M(x, R_M, \beta^{(1)}, \ldots \beta^{(i-1)})$. We denote by $\text{conv}_M(x, R_M, \beta^{(1)}, \ldots, \beta^{(t)}) = (x, \alpha, \beta, \gamma)$ the conversation output by the simulator M when activated with these parameters (notice that t strings $\beta^{(i)}$ always suffice for answering all tries of M). By our convention on the simulator M, there exists i, $1 \leq i \leq t$, such that $\alpha = \alpha^{(i)}$ and $\beta = \beta^{(i)}$.

DEFINITION. *We say that a vector* $(x, R_M, \beta^{(1)}, \ldots, \beta^{(t)})$ *is M-good if* $\text{conv}_M(x, R_M, \beta^{(1)}, \ldots, \beta^{(t)})$ *is an accepting conversation for the (honest) verifier V, namely, if* $\text{conv}M(x, R_M, \beta^{(1)}, \ldots, \beta^{(t)}) = (x, \alpha, \beta, \gamma)$ *and* $\rho_V(x, \alpha, \beta, \gamma) = ACCEPT$. *We say that* $(x, R_M, \beta^{(1)}, \ldots, \beta^{(t)})$ *is (M, i)-good (or i-good for short) if it is M-good and* $\alpha = \alpha^{(i)}$, $\beta = \beta^{(i)}$.

The main property of M-good strings is stated in the following lemma.

LEMMA 6.3. *Let* $\langle P, V\rangle$ *be a three-round Arthur–Merlin protocol for a language L. Suppose* $\langle P, V\rangle$ *is black-box simulation zero-knowledge, and let M be a black-box simulator as above. Then,*

1. *for strings x outside L, only a negligible portion of the vectors* $(x, R_M, \beta^{(1)}, \ldots, \beta^{(t)})$ *are M-good*;
2. *for strings x in L there exists a nonnegligible portion of the vectors* $(x, R_M, \beta^{(1)}, \ldots, \beta^{(t)})$ *that are M-good. (This nonnegligible portion is at least one half of the completeness probability of the protocol* $\langle P, V\rangle$*, i.e., at least half the probability that P convinces V to accept x.)*

Before proving this key lemma, we use it to prove Theorem 6.1 for the case of the three-round Arthur–Merlin interactive proof.

Proof of Theorem 6.1 (*for the case* AM(3)). By Lemma 6.3 we get the following BPP algorithm for the language L. On input x:

*select at random a vector $(R_M, \beta^{(1)}, \ldots, \beta^{(t)})$;

*accept x if and only if $(x, R_M, \beta^{(1)}, \ldots, \beta^{(t)})$ is M-good.

The complexity of this algorithm is like the complexity of testing for M-goodness. The latter is polynomial-time since it involves running the simulator M which is polynomial-time, and evaluating the predicate ρ_V, which is also polynomial-time computable. The success probability of the algorithm is given by Lemma 6.3. □

Proof of Lemma 6.3. (1) Assume that the portion of M-good vectors $(x, R, \beta^{(1)}, \ldots, \beta^{(t)})$ for x's not in L is not negligible. This means that there exist infinitely many $x \notin L$ for which the portion of M-good vectors is nonnegligible. For each such x, there exists an index i_0, $1 \leq i_0 \leq t$, for which a nonnegligible fraction of the vectors $(x, R, \beta^{(1)}, \ldots, \beta^{(t)})$ are i_0-good (since there are only polynomially many possible values for i_0). Thus, there exists a nonnegligible number of prefixes $(x, R, \beta^{(1)}, \ldots, \beta^{(i_0-1)})$, each with a nonnegligible number of i_0-good continuations $(\beta^{(i_0)}, \ldots, \beta^{(t)})$ (i.e., such that $(x, R, \beta^{(1)}, \ldots, \beta^{(i_0-1)}, \beta^{(i_0)}, \ldots, \beta^{(t)})$ are i_0-good). Let $(x, R, \beta^{(1)}, \ldots, \beta^{(i_0-1)})$ be such a prefix, and let $\alpha^{(i_0)} = \alpha_M(x, R, \beta^{(1)}, \ldots, \beta^{(i_0-1)})$. For each i_0-good continuation $(\beta^{(i_0)}, \ldots, \beta^{(t)})$ machine M outputs a conversation $(x, \alpha^{(i_0)}, \beta^{(i_0)}, \gamma)$ for which $\rho_V(x, \alpha^{(i_0)}, \beta^{(i_0)}, \gamma) = ACCEPT$. In particular, there exists a nonnegligible number of $\beta^{(i_0)}$ for which this happens.

In other words, for each x as above, there exists a string $\alpha_x (= \alpha^{(i_0)})$ for which the set $B(x, \alpha_x) = \{\beta : \exists\gamma, \rho_V(x, \alpha_x, \beta, \gamma) = ACCEPT\}$ is of nonnegligible size among all possible strings β. Consider now a ("cheating") prover that sends this α_x as its first message. If V responds with $\beta \in B(x, \alpha_x)$, the prover sends the corresponding γ, which convinces V to accept. Since V selects its messages β at random, then the probability of being convinced by the above prover is (at least) as big as the relative size of $B(x, \alpha_x)$, i.e., nonnegligible. Concluding, we have shown the existence of a prover that for infinitely many x's outside L convinces V to accept with nonnegligible probability. This contradicts the soundness condition of the protocol $\langle P, V\rangle$, and this part of the lemma follows.

(2) We show that for strings x in L a nonnegligible portion of the vectors $(x, R_M, \beta^{(1)}, \ldots, \beta^{(t)})$ are M-good. We do it by considering the behavior of the simulator M when receiving "random-like" responses from the verifier. This behavior is analyzed by introducing a particular family of "cheating" verifiers, each of them associated to a different hash function from a family of *$t(n)$-wise independent hash functions*. The $t(n)$-wise independence (where $t(n)$ is the bound on the number of simulator's tries) achieves the necessary randomness from the verifiers' responses.

Let $x \in L$ and let n denote its length. Consider a family of hash functions H_n which map $l(n)$-bit strings into $l(n)$-bit strings, such that the locations assigned to the strings by a randomly selected hash function are uniformly distributed and $t(n)$-wise independent. (Recall that $l(n)$ is the length of messages α and β in the Arthur–Merlin protocol $\langle P, V\rangle$ for L, while $t(n)$ is the bound on the number of M's tries.) For properties and implementation of such functions, see, e.g., [Jof], [WC], and [CG]; in particular, we observe that such functions can be described by a string of length $t(n) \cdot l(n)$, i.e., polynomial in n.

For each hash function $h \in H_n$ we associate a (deterministic nonuniform) verifier V_h^*, which responds to the prover's message α with the string $\beta = h(\alpha)$ (V_h^* has wired in the description of h). Consider the simulation of $\langle P, V_h^*\rangle$ conversations by the simulator M. Fixing an input x, a random tape R_M for M, and a function $h \in H_n$, the whole simulation is determined. In particular, this (uniquely) defines a sequence of α's tried by the simulator, and the corresponding responses β of V_h^*. We denote by $\alpha^{(1)}, \alpha^{(2)}, \ldots, \alpha^{(s)}$, the *different* values of α in these tries. When $s < t$, we complete this sequence to $\alpha^{(1)}, \ldots, \alpha^{(s)}, \alpha^{(s+1)}, \ldots, \alpha^{(t)}$, by adding $t - s$ strings α in some canonical way, such that the resultant $\alpha^{(1)}, \ldots, \alpha^{(t)}$ are all different. Let $\beta^{(i)} = h(\alpha^{(i)})$, $1 \le i \le t$, and define $v(x, R_M, h) = (x, R_M, \beta^{(1)}, \ldots, \beta^{(t)})$. Part (2) of the lemma follows from the following two claims.

CLAIM 1. For $x \in L$, there is a nonnegligible portion of the pairs (R_M, h) for which the vector $v(x, R_M, h)$ is M-good.

Proof. For any input x to the protocol $\langle P, V\rangle$, let p_x denote the probability that the prover P convinces V (the honest verifier) to accept x. In other words, p_x is the probability, over the coin sequences R_P of the prover P, and (random) choices β of V, that the resultant conversation $(x, \alpha(x, R_P), \beta, \gamma(x, R_P, \beta))$ is accepting. By the completeness property of the protocol $\langle P, V\rangle$, we get that for x's in L the probabilities p_x are nonnegligible.

Let $x \in L$ and consider the interaction between the real prover P and the verifiers V_h^* on the input x. Each coin sequence R_P determines the message α and the corresponding response $h(\alpha)$ by V_h^*. By the uniformity property of the family H_n we get that for every α, all β's are equiprobable as the result of $h(\alpha)$. Therefore, the probability that P and V_h^* (for h uniformly chosen from H_n) output an accepting conversation is exactly the same as the probability, p_x, that P and V output such a conservation.

Finally, since the simulator M succeeds in simulating $\langle P, V_h^*\rangle$ conversations for all functions $h \in H_n$, we get that for each h the probability that M outputs an accepting conversation when interacting with V_h^* is almost the same (up to negligible difference) as the probability that P and V_h^* output an accepting conversation. This last probability, for $h \in_R H_n$, is p_x. We conclude that the probability, over random R_M and h, that $v(x, R_M, h)$ is M-good is almost p_x and thus nonnegligible. The claim follows. □

CLAIM 2. For all strings x and R_M, and for h chosen with uniform probability from H_n, the vector $v(x, R_M, h)$ is uniformly distributed over the set $\{(x, R_M, \beta^{(1)}, \ldots, \beta^{(t)}) : \beta^{(i)} \in \{0, 1\}^{l(n)}\}$.

Proof. Recall the function α_M introduced above. Observe that

$$v(x, R_M, h) = (x, R_M, \beta^{(1)}, \ldots, \beta^{(t)})$$

if and only if for every i, $1 \le i \le t$,

$$h(\alpha_M(x, R_M, \beta^{(1)}, \ldots, \beta^{(i-1)})) = \beta^{(i)}.$$

On the other hand, by the uniformity and $t(n)$-independence property of the family H_n, we have that for any t *different* elements $a_1, \ldots, a_t$ in the domain of the functions $h \in H_n$, the sequence $h(a_1), \ldots, h(a_t)$ is uniformly distributed over all the possible sequences $b_1, \ldots, b_t$ for b_i in the range of the functions H_n.

Thus, for all strings x and R_M, and for fixed $\beta^{(1)}, \ldots, \beta^{(t)}$, the probability (for $h \in_R H_n$) that $v(x, R_M, h) = (x, R_M, \beta^{(1)}, \ldots, \beta^{(t)})$ equals the probability that for every i, $1 \leq i \leq t$, h maps $\alpha^{(i)} = \alpha_M(x, R_M, \beta^{(1)}, \ldots, \beta^{(i-1)})$ into $\beta^{(i)}$. Since, by definition, all $\alpha^{(i)}$'s are different, then we can use the above property of the family H_n to get that the latter probability is the same for every sequence $\beta^{(1)}, \ldots, \beta^{(t)}$ (i.e., we put $a_i = \alpha^{(i)}$ and $b_i = \beta^{(i)}$). The claim follows. □

Claim 2 states that for any R_M, the value of $v(x, R_M, h)$ is uniformly distributed over all possible vectors $(x, R_M, \beta^{(1)}, \ldots, \beta^{(t)})$. On the other hand, by Claim 1, a nonnegligible portion of $v(x, R_M, h)$ are M-good, and then we get that a nonnegligible portion of the vectors $(x, R_M, \beta^{(1)}, \ldots, \beta^{(t)})$ are M-good.

The lemma follows. □

Remark 6.1 (*expected polynomial-time simulator*). For simplicity we have assumed that the given simulator, M, for the protocol $\langle P, V\rangle$ runs in (strictly) polynomial time. Nevertheless, in the definition of zero-knowledge we allow this simulator to run in *expected* polynomial time. We show that our results also hold in this general case by transforming a given expected polynomial-time simulator M into a strictly polynomial-time simulator M', and showing that Lemma 6.3 holds for this new simulator. Then, we can use the modified simulator M' in the BPP algorithm for the language L.

The simulator M' behaves like M, but its running time is truncated after some (fixed) polynomial number of steps, denoted $s(n)$. We show how to choose this polynomial $s(n)$. Let $T(n)$ be a polynomial bounding the *expected* running time of M, and let $p(n)$ be a (lower) bound on the probability that the prover P convinces the (honest) verifier V to accept an input in L of length n. We define $s(n)$ to be $2 \cdot T(n)/p(n)$. Since $1/p(n)$ is polynomially bounded (by the completeness condition of the protocol $\langle P, V\rangle$), then $s(n)$ is polynomially bounded. With this modification of M the proof of Lemma 6.3 remains valid, except for a more delicate argument in the proof of Claim 1. The required changes follow.

In that proof we claimed that "for each h the probability that M outputs an accepting conversation when interacting with V_h^* is almost the same (up to a negligible difference) as the probability that P and V_h^* output an accepting conversation." This is true for the original simulator M, but not necessarily for M'. Since we cut the running of M after $s(n)$ steps, then there exist cases in which M' does not complete the original behavior of M. Nevertheless, by the choice of $s(n)$, the probability (over the coin tosses of M') that this happens (i.e., the running time of M exceeds $s(n)$) is at most $p(n)/2$. Thus, for any h, the probability that the truncated simulator, M', outputs an accepting conversation when interacting with V_h^* differs from the probability that P and V_h^* output an accepting conversation by at most $p(n)/2$. For $h \in_R H_n$, this last probability was shown (in the original proof of Claim 1) to be at least $p(n)$, and then we get that the probability, over random $R_{M'}$ and h, that $v(x, R_{M'}, h)$ is M'-good is (up to a negligible difference) larger than $p(n)/2$, and then nonnegligible. Therefore, Claim 1 follows in this case also. □

Remark 6.2 (*randomized* ρ_V). We have assumed that the only coin tosses of the (honest) verifier V during the Arthur–Merlin protocol $\langle P, V\rangle$ are the bits corresponding to the string β sent to the prover, and that no additional coin tosses are used in order to compute the accepting/rejecting predicate ρ_V. This restriction can be removed from the above proof by using finer arguments, as done in our treatment of the general IP(3) case (of §6.3).

More generally, any AM(k) protocol in which the predicate ρ_V depends on the whole conversation and some additional random string can be transformed into an AM($k + 1$) protocol in which no such additional string is used: simply let the verifier send this random string as its last message. Hence, since we prove our result for any constant-round AM protocol, we can assume that ρ_V is deterministic. □

Remark 6.3 (*interactive arguments*). We now show how to generalize the above proof of the case AM(3) in order to prove the same result in the setting of interactive arguments, i.e., "interactive proofs" in which the soundness condition is required only with respect to provers limited to probabilistic polynomial-time computations, possibly getting an auxiliary input. We have to prove Lemma 6.3 in this setting. Notice that part (2) of the lemma relies on the completeness and zero-knowledge properties of the interactive proof, but these properties are not influenced by the soundness condition. Therefore, this part of the proof automatically holds for interactive arguments. The other part, part (1), relies on the soundness of the interactive proof, thus a modification is required in the proof to deal with provers having just polynomial power.

In that proof we showed, by contradiction, the existence of infinitely many x's not in L for which a cheating prover can convince the verifier to accept x with nonnegligible probability. The success of this prover was shown by proving, for each such x, the *existence* of a message α_x that for nonnegligibly many β's a string γ exists such that $\rho_V(x, \alpha_x, \beta, \gamma) = ACCEPT$. In the interactive arguments, setting the sole existence of such an α_x is not sufficient. The limited prover should find in probabilistic polynomial time this string and the corresponding response γ to the message β sent by V. We describe such a prover P^*, which uses the simulator M in order to find the required strings. It begins by choosing $i \in_R \{1, \dots, t\}$ and random strings R_M, $\beta^{(1)}, \dots, \beta^{(i-1)}$. Then it computes $\alpha = \alpha_M(x, R_M, \beta^{(1)}, \dots, \beta^{(i-1)})$ and sends this α to V. Once V responds with β, the prover P^* chooses $t-i$ random strings $\beta^{(i+1)}, \dots, \beta^{(t)}$, computes (using the simulator M) the conversation $\mathrm{conv}_M(x, R_M, \beta^{(1)}, \dots, \beta^{(i-1)}, \beta, \beta^{(i+1)}, \dots, \beta^{(t)})$, and sends to V the message γ appearing in this conversation. If the chosen vector is i-good then this γ convinces V to accept the conversation. We analyze the probability of such an event.

There exists a nonnegligible probability that P^* chooses i, $1 \leq i \leq t$, for which the number of i-good vectors is nonnegligible (we saw that such an i exists). On the other hand, the whole vector $(x, R_M, \beta^{(1)}, \dots, \beta^{(i-1)}, \dots, \beta^{(t)})$ is chosen at random (except for x): the β component by the verifier (the protocol is Arthur–Merlin!) and the other components by P^*. Therefore, there is a nonnegligible probability that the resultant vector is i-good, in which case V accepts x. This way P^* works in polynomial time and has a nonnegligible probability of convincing V to accept x, from which we derive the required contradiction. □

6.2. The case AM(k): Secret coins help zero-knowledge. In this section we consider constant-round Arthur–Merlin interactive proofs. We show that a language having such an interactive proof which is also black-box simulation zero-knowledge belongs to BPP, thus proving Theorem 6.1. We present this proof based on the proof for the particular case of AM(3) as given in §6.1. The basic ideas are similar, but their implementation is technically more involved in this general case. We highly recommend familiarity with §6.1 before going through the present section.

The protocol ⟨P, V⟩. Let $\langle P, V\rangle$ be a k-round Arthur–Merlin protocol for a language L. For simplicity of the exposition we make some assumptions on the form of the protocol without restricting the generality of the proof. We consider protocols in which both the first and last messages are sent by the prover. By adding dummy messages any protocol can be converted into one of this form. Notice that in such a protocol, the number of rounds is always an odd number $k = 2 \cdot m + 1$. The prover P sends $m + 1$ messages which we denote by $\alpha_1, \dots, \alpha_m$ and γ, respectively. The m messages by V are denoted $\beta_1, \dots, \beta_m$. The input to the protocol is denoted by x, and its length by n. The predicate computed by the verifier V in order to accept or reject the input x is denoted by ρ_V, and we assume it to be a deterministic function of the conversation $\rho_V(x, \alpha_1, \beta_1, \ldots, \alpha_m, \beta_m, \gamma)$. (Our results hold also for interactive proofs in which ρ_V depends on an additional random string. See Remark 6.2.) We need the following technical convention. We assume that all prover messages in the protocol have a form that allows them, by only seeing the ith message α_i, to uniquely reconstruct all previous messages sent by the prover during the conversation. This is easily achieved by simple concatenation of previous messages (using a delimiter or some length convention). We also assume the existence of a polynomial $l(n)$ such that all prover's and verifier's messages on an n-length input have length $l(n)$ (e.g., using dummy padding). Finally, we let the verifier V check whether the received messages conform to the above conventions, and reject the conversation if not.

The simulation process. We denote by M the black-box simulator for the protocol $\langle P, V \rangle$. The simulation process consists of several tries by the simulator M. Each try involves feeding the verifier V^* (i.e., the black box representing it) with a value y as the input to the protocol, and the messages α_i, $1 \leq i \leq m$, that simulate the messages sent by P. (Again, we do not care about random coins for V^*; we just need a simulator that is able to simulate conversations with deterministic verifiers.) The simulator M chooses these arguments, in the successive tries, depending on the random tape R_M and the responses β_i output by the black box V^* during the current and previous tries. After each try the simulator may decide to output a conversation of the form $(y, \alpha_1, \beta_1, \ldots, \alpha_m, \beta_m, \gamma)$ or to perform a new try. We assume that the output conversation has $y = x$ (i.e., the input component in the conversation corresponds to the actual input being simulated), that the α messages appearing in the output conversation fit our convention on the form of the prover's messages, and that the simulator explicitly tries the output conversation. Namely, it operates (in one of the tries) the black box V^* on input x and $\alpha_1, \ldots, \alpha_m$ as appearing in the output conversation, and, respectively, gets as responses to V^* the strings $\beta_1, \ldots, \beta_m$, also appearing in this conversation. These assumptions are apparently restricting ones, since the simulator is allowed to output conversations that are not "legal conversations" between the prover P and the simulated verifier V^*. Nevertheless, a simulator that succeeds simulating the $\langle P, V^* \rangle$ conversations will output such illegal conversations with only negligible probability (otherwise the simulated conversations can be easily distinguished from the true ones). Finally, we consider, for the sake of simplicity, only simulators that run in (strictly) polynomial time. The necessary changes in the proof for handling the general case in which the simulator runs in expected polynomial time are analogous to the ones described in Remark 6.1 for the case AM(3). We denote by $\hat{t}(n)$ a polynomial bounding the number of calls to V^* tried by M before outputting a conversation, and put $t(n) = m \cdot \hat{t}(n)$ (notice that $t(n)$ constitutes an upper bound on the total number of messages α tried by M during the whole simulation).

The simulator as a subroutine. Our goal is to present a BPP algorithm for the language L, and we use the simulator M to achieve it. The way M is used is similar to the way we used the simulator in the AM(3) case (see §6.1). In the present case, the behavior of the simulator M when "interacting" with a verifier V^* is determined by the input x to the protocol, the random tape R_M, and the strings β output by V^* as responses to the strings fed by M during the different tries. Also, here we define a computational process that uses M as a subroutine.

Fix an input x of length n, a string R_M, and $t = t(n)$ strings $\beta^{(1)}, \beta^{(2)}, \ldots, \beta^{(t)}$, each of length $l(n)$. Activate M on input x with its random tape containing R_M. For each message α presented by M, respond in the following way. (The responses will depend on the strings α, but not on y.) If α is "illegal," then respond with a special "reject-message." By illegal we mean a message α that does not fit our above conventions on the form of the prover's messages. For legal α's we respond (impersonating a black-box verifier) with one of the β's from the above list $\beta^{(1)}, \ldots, \beta^{(t)}$ according to the following rule. If the same α was previously presented by M (i.e., during a previous try), respond with the same β as in that case. If α is the ith *different* (legal) string presented by M since the beginning of the simulation, then respond with $\beta^{(i)}$. We denote the ith different α by $\alpha^{(i)}$. Clearly, $\alpha^{(i)}$ is uniquely determined by x, R_M, and the $i-1$ strings $\beta^{(1)}, \ldots, \beta^{(i-1)}$. That is, there exists a deterministic function α_M such that $\alpha^{(i)} = \alpha_M(x, R_M, \beta^{(1)}, \ldots, \beta^{(i-1)})$. We denote by $\mathrm{conv}_M(x, R_M, \beta^{(1)}, \ldots, \beta^{(t)}) = (y, \alpha_1, \beta_1, \ldots, \alpha_m, \beta_m, \gamma)$ the conversation output by the simulator M when activated with these parameters (notice that t strings $\beta^{(i)}$ always suffice for answering all tries of M). By our convention on the simulator M and on the form of the prover's messages it follows that there exists a sequence of indices $1 \leq i_1 < i_2 < \cdots < i_m \leq t$ such that for each α_j, β_j, $j = 1, \ldots, m$, appearing in the output conversation, $\alpha_j = \alpha^{(i_j)}$ and $\beta_j = \beta^{(i_j)}$. This is true since the simulator always outputs a conversation which was explicitly generated in one of its tries. The increasing property of the sequence of indices i_j is enforced by the special form of the "legal" messages α, namely, by the fact that we respond to message α_j only if we had previously responded to the messages $\alpha_1, \ldots, \alpha_{j-1}$. In the present setting we use the following definition of M-good vectors.

DEFINITION. *We say that a vector* $(x, R_M, \beta^{(1)}, \ldots, \beta^{(t)})$ *is M-good if* $\mathrm{conv}_M(x, R_M, \beta^{(1)}, \ldots, \beta^{(t)})$ *is an accepting conversationfor the (honest) verifier V. We say that* $(x, R_M, \beta^{(1)}, \ldots, \beta^{(t)})$ *is* $(i_1, i_2, \ldots, i_m)$*-good if it is M-good and the corresponding conversation has* $\alpha_j = \alpha^{(i_j)}$ *and* $\beta_1 = \beta^{(i_j)}$*, for* $j = 1, \ldots, m$.

The following lemma is analogous to Lemma 6.3.

LEMMA 6.4. *Let* $k = 2 \cdot m + 1$ *be a constant, and let* $\langle P, V\rangle$ *be a k-round Arthur–Merlin protocol for a language L. Suppose* $\langle P, V\rangle$ *is black-box simulation zero-knowledge, and let M be a black-box simulator as above. Then*

1. *for strings x outside L, only a negligible portion of the vectors* $(x, R_M, \beta^{(1)}, \ldots, \beta^{(t)})$ *are M-good*;
2. *for strings x in L there exists a nonnegligible portion of the vectors* $(x, R_M, \beta^{(1)}, \ldots, \beta^{(t)})$ *that are M-good. (This nonnegligible portion is at least one half of the completeness probability of the protocol* $\langle P, V\rangle$*, i.e., half the probability that P convinces V to accept x).*

Proof of Theorem 6.1. Using Lemma 6.4 we get that the algorithm described in the proof of Theorem 6.1 for the special case of AM(3) (see §6.1) is a BPP algorithm for the language L. □

Proof of Lemma 6.4. This proof is essentially analogous to the proof of Lemma 6.3, although some delicate modifications are required.

(1) Assume that the portion of M-good vectors $(x, R_M, \beta^{(1)}, \ldots, \beta^{(t)})$ for x's not in L is not negligible. This means that there exist infinitely many $x \notin L$ for which the portion of M-good vectors is nonnegligible. Observe that there are only polynomially many different sequences $1 \leq i_1 < i_2 < \cdots < i_m \leq t$ (i.e., $\binom{t(n)}{m}$, and m is a constant), and then note that for each x, as above, there exists a sequence $(i_1, i_2, \ldots, i_m)$ for which nonnegligibly many vectors $(x, R_M, \beta^{(1)}, \ldots, \beta^{(t)})$ are $(i_1, i_2, \ldots, i_m)$-good. Next, we describe a prover P^* which convinces the (honest) verifier V to accept any of the above inputs $x \notin L$ with nonnegligible probability, thus contradicting the soundness condition of the protocol $\langle P, V\rangle$.

The prover P^* begins by choosing a sequence $(i_1, i_2, \ldots, i_m)$ at random. Then, it chooses random strings $R_M, \beta^{(1)}, \ldots, \beta^{(i_1-1)}$ and uses them to compute $\alpha_1 = \alpha_M(x, R_M, \beta^{(1)}, \ldots, \beta^{(i_1-1)})$. It sends α_1 to V and receives back the response β_1. Now P^* chooses random $\beta^{(i_1-1)}, \ldots, \beta^{(i_2-1)}$ and computes $\alpha_2 = \alpha_M(x, R_M, \beta^{(1)}, \ldots, \beta^{(i_1-1)}, \beta_1, \beta^{(i_1+1)}, \ldots, \beta^{(i_2-1)})$. After receiving the response β_2 from the verifier, P^* selects new random strings $\beta^{(i_2+1)}, \ldots, \beta^{(i_3-1)})$ and computes $\alpha_3 = \alpha_M(x, R_M, \beta^{(1)}, \ldots, \beta^{(i_1-1)}, \beta_1, \beta^{(i_1+1)}, \ldots, \beta^{(i_2-1)}, \beta_2, \beta^{(i_2+1)}, \ldots, \beta^{(i_3-1)})$. This process continues until all messages α_i, β_i, $1 \leq i \leq m$, are computed and exchanged. When the resultant vector $(x, R_M, \beta^{(1)}, \ldots, \beta^{(i_1-1)}, \beta_1, \beta^{(i_1+1)}, \ldots,$

$\beta^{(i_2-1)}, \beta_2, \ldots, \beta^{(t)})$ is $(i_1, i_2, \ldots, i_m)$-good, then computing the function conv_M on this vector results in an accepting (for V) conversation $(x, \alpha_1, \beta_1, \ldots, \alpha_m, \beta_m, \gamma)$ (with α_i, β_i, as defined above). But then, by sending this γ, the prover P^* convinces V to accept. The probability that this happens equals the probability that the above vector $(x, R_M, \beta^{(1)}, \ldots, \beta^{(i_1-1)}, \beta_1, \beta^{(i_1+1)}, \ldots, \beta^{(i_2-1)}, \beta_2, \ldots, \beta^{(t)})$ is $(i_1, i_2, \ldots, i_m)$-good. Since this sequence of indices and all the vector components (excluding x) are chosen at random (recall that V chooses its messages, $\beta_1, \ldots, \beta_m$, at random!) then this probability is nonnegligible.

(2) The proof of this part is analogous to the corresponding proof in Lemma 6.3. We use a set H_n of $t(n)$-independent hash functions ($t(n)$ as defined in this section) to define a family of verifiers V_h^*. For all $h \in H_n$, the verifier V_h^* responds to a legal message α sent by the prover with $h(\alpha)$, and with a rejection message if α is illegal. The statements for Claims 1 and 2 remain the same, as does the proof of Claim 2. The proof of Claim 1 needs a more delicate argument, as follows. As in the AM(3) case we consider the interaction between the prover P and a verifier V_h^*, but now this interaction generates a conversation of the form $(x, \alpha_1, \beta_1, \ldots, \alpha_m, \beta_m, \gamma)$. In particular, for each h and random tape R_P for P a unique sequence of messages $\beta_1, \ldots, \beta_m$ (the responses of V_h^*) is determined. We have to show that for every tape R_P all sequences $\beta_1, \ldots, \beta_m$ are equiprobable for $h \in_R H_n$. The proof of this property uses a similar argument as in the proof of Claim 2: observe that the pair R_P and h generates the responses $\beta_1, \ldots, \beta_m$ if and only if, for every i, $1 \le i \le m, h(\alpha_P(x, R_P, \beta_1, \ldots, \beta_{i-1})) = \beta_i$. (Here α_P stands for the function computed by P in order to determine its next message α.) Thus, the probability (for $h \in_R H_n$) that a given sequence $\beta_1, \ldots, \beta_m$ is generated is like the probability that for every i, $1 \le i \le m, h$ maps $\alpha_P(x, R_P, \beta_1, \ldots, \beta_{i-1})$ into β_i. Since the functions H_n are m-independent (by definition they are $t(n)$-independent, but $m \le t(n)$), and the messages $\alpha_1, \ldots \alpha_m$ output by P are all different by convention, we get that the latter probability is the same for every sequence $\beta_1, \ldots, \beta_m$.

From this property of the pairs R_P and h we conclude that the probability that P and V_h^* (for $h \in_R H_n$) output an accepting conversation is exactly the same as the probability that P and the honest V output such a conversation.

The rest of the proof follows as in Claim 1 of Lemma 6.3 □

Remark 6.4. Notice that the prover P^* described in the proof of part (1) of Lemma 6.4 is a polynomial-time prover. The other parts of the proof of Theorem 6.1 also hold for such provers, and then we get that our result remains valid also in the setting of interactive arguments.

6.3. The case IP(3). In the setting of general interactive proofs the (honest) verifier is not restricted to choosing all its messages at random, but can compute them based on the input x, a random ("secret") string r, and the previous messages of the prover. In the case of three rounds this means that the only message sent by V during the protocol is computed by means of a (deterministic) function $\beta_V(x, r, \alpha)$, where α is the first message sent by P. Also, in this case V accepts or rejects a conversation based on a predicate $\rho_V(x, r, \alpha, \gamma)$ (γ is the last message sent by P).

Here we outline the proof of Theorem 6.2, based on the proof presented in §6.1 for the AM(3) case.

Proof of Theorem 6.2 (*outline*). Let L be a language having a three-round interactive proof which is black-box simulation zero-knowledge. Let $\langle P, V\rangle$ be such a protocol and let M be the corresponding black-box simulator. The simulation process consists of several tries; in each of them the simulator feeds the black box V^* with arguments y (the input) and α (the prover's message), and gets an answer β from V^*. (Again, it suffices to consider a simulator just able to simulate conversations with deterministic verifiers, so this simulator does not feed V^* with a random tape.) We assume the same conventions on the simulator as the ones described in §6.1 for the proof of the AM(3) case.

- The simulator always outputs a conversation of the form $(x, \alpha, \beta, \gamma)$, where x and α are fed into the black-box V^* in one of the simulator tries, and β is the response of V^* to these arguments.
- The simulator runs in strictly polynomial time. In particular, $t(n)$ stands for the polynomial bound on the number of tries made by M on inputs of length n during the simulation process (the case of expected polynomial-time simulators is handled exactly as in Remark 6.1).

The main modification with respect to the proof of the AM(3) case is in the way we use the simulator M as a subroutine for constructing the BPP algorithm for the language L. Recall that the whole simulator process is completely determined by the input to the protocol, x, the contents of M's random tape, R_M, and the responses by the verifier. This was true for the AM(3) case and remains true here. In the former case we used M as a subroutine by feeding it with x and a randomly chosen string R_M. Then, we used $t = t(n)$ random strings $\beta^{(1)}, \ldots, \beta^{(t)}$ as the response of the virtual verifier. In the present case we choose a string R_M, as before, and t random strings denoted $r^{(1)}, \ldots, r^{(t)}$, each of length $l(n)$, where $l(n)$ is a (polynomial) bound on the number of random bits used by the (honest) verifier in the IP(3) protocol $\langle P, V\rangle$. The idea is to use these strings as the random coins of the virtual verifier for responding to M's tries. More precisely, for each try by the simulator, consisting of an input y to the protocol and a message α, we compute $\beta = \beta_V(y, r^{(i)}, \alpha)$, and feed it into M as the verifier's response to α. For each new try we use a new $r^{(i)}$ (in increasing order of i), except in the case in which the present α was also presented in a previous try. If so, we use the same $r^{(i)}$ as in that case.

Note that a unique conversation is determined by x, R_M, and the t strings $r^{(1)}, \ldots, r^{(t)}$. Thus, as in the case AM(3), we can define $\mathrm{conv}_M(x, R_M, r^{(1)}, \ldots, r^{(t)})$ to be the conversation output by M when the described process is finished. Also, we denote by $\alpha^{(i)}$ the ith *different* α output by M during the simulation. Clearly, $\alpha^{(i)}$ is uniquely determined by x, R_M, and the strings $r^{(1)}, \ldots, r^{(i-1)}$; thus we denote $\alpha^{(i)} = \alpha_M(x, R_M, r^{(1)}, \ldots, r^{(i-1)})$.

By our convention on M, if $\mathrm{conv}(x, R_M, r^{(1)}, \ldots, r^{(t)}) = (x, \alpha, \beta, \gamma)$, then M explicitly tried the arguments x and α during the simulation, and got as response the string β. This means that there exists (at least one) i, $1 \leq i \leq t$, such that $\beta = \beta_V(x, r^{(i)}, \alpha)$. This fact is used in the following definition.

DEFINITION. *We say that a vector* $(x, R_M, r^{(1)}, \ldots, r^{(t)})$ *is M-good if* $\mathrm{conv}(x, R_M, r^{(1)}, \ldots, r^{(t)}) = (x, \alpha, \beta, \gamma)$ *and* $\rho_V(x, r^{(i)}, \alpha, \gamma) =$ *ACCEPT, where i is the minimal value for which* $\beta = \beta_V(x, r^{(i)}, \alpha)$. *According to this value of i, we call the conversation (M, i)-good (or i-good for short).*

Using this re-definition of the notion of M-goodness, Lemma 6.3 of §6.1 also holds in the present (IP(3)) case, by just changing the $\beta^{(i)}$ notation by $r^{(i)}$ in the formulation of the lemma. Theorem 6.2 then follows by using the BPP algorithm as described in the proof of the AM(3) case. For the proof of Lemma 6.3 in the present case we note the following simple modifications. In the proof of part (1) we use the same reasoning as in the corresponding proof in §6.1 but applied to the strings $r^{(i)}$ instead of $\beta^{(i)}$. We note that the soundness probability of the protocol is now defined over the random coins used by V, i.e., over the choices $r^{(i)}$. For the proof of the other part of the lemma we slightly modify the definition of the verifiers V_h^*. We still use the same family of hash functions, but the verifier V_h^* works as follows: on message α sent by the prover, V_h^* responds with $\beta = \beta_V(x, h(\alpha), \alpha)$, i.e., it computes β as the honest verifier does, but using $h(\alpha)$ as the random coins of V. The rest of the proof (including Claims 1 and 2) remains essentially unchanged (up to the replacement of "responses $\beta^{(i)}$" by "random coins $r^{(i)}$"). □

Remark 6.5. As in the previous cases also, the IP(3) case extends to the setting of interactive arguments. The modifications in the proof are analogous to the ones described in Remark 6.3.

7. Concluding remarks. Although the results presented in this paper are negative in nature, we believe that they have played a positive role in the development of the field.

We believe that sequential composition is a fundamental requirement of zero-knowledge protocols. It is analogous to requiring that adding two algebraic expressions, each evaluating to zero, yields an expression which evaluates to zero. Furthermore, sequential composition is required when using zero-knowledge proofs as tools in the design of cryptographic protocols (an application which is the primary motivation of zero-knowledge). *The fact that the original formulation of zero-knowledge is not closed under sequential composition establishes the importance of augmenting this formulation by an auxiliary input* (cf. [GO], [Ore], [TW], and [Gol]). It should be stressed, of course, that all known zero-knowledge proofs also satisfy the augmented formulation.

Parallel composition is the key to improving the efficiency (in terms of number of rounds) of zero-knowledge protocols, but we do not believe that it is a fundamental requirement. Carrying the analogy of the previous paragraph, one cannot require that "interleaving" two expressions (each evaluating to zero) yields an expression which evaluates to zero. *The fact that all known formulations of (computational) zero-knowledge are not closed under parallel composition motivates the introduction of weaker notions such as* witness indistinguishability [FS2] *which suffice for many applications.* Namely, instead of strengthening the hypothesis of the alleged Parallel Composition Theorem (as was done in the case of Sequential Composition), one relaxes the conclusion of the Parallel Composition Theorem (and this weaker conclusion turns to suffice in many applications).

The fact that ("nontrivial") black-box zero-knowledge proofs cannot be both of AM type and of constant number of rounds establishes the importance of "private coins" in the design of constant-round zero-knowledge proofs. In other words, in the process of such proofs, the verifier must "commit" (and later "decommit") to some pieces of information. In fact, such commitments are the core of the constant-round zero-knowledge proofs (and arguments) for any language in NP presented in [BCY], [FS1], and [GKa] (relying on various reasonable intractability assumptions) and in the (unconditional) zero-knowledge proof for graph isomorphism presented in [BMO1].

Appendix: Proof of existence of P/poly-evasive pseudorandom ensembles. In this appendix we present the proof of Theorem 3.2. We first restate the theorem.

THEOREM 3.2. *There exists a* P/poly*-evasive ensemble* $\boldsymbol{S}^{(1)}, \boldsymbol{S}^{(2)}, \ldots$ *with* $Q(n) = 4n$, *such that for every* n, *each* $S_i^{(n)}$ *is a* $(2^{n/4}, 2^{-n/4})$*-pseudorandom set of cardinality* 2^n. *Furthermore, there exists a Turing machine which on input* 1^n *outputs the collection* $\boldsymbol{S}^{(n)}$.

Proof. For any integer n, we denote by $\boldsymbol{R}^{(n)}$ the collection of sets $S \subseteq \{0, 1\}^{4n}$ of cardinality 2^n which are $(2^{n/4}, 2^{-n/4})$-pseudorandom, and by $\boldsymbol{C}^{(n)}$ the set of (deterministic) circuits of size $2^{n/4}$ having n inputs and $4n$ outputs.

We prove the theorem by showing, for any large enough n, the existence of 2^n sets $S_1, \ldots, S_{2^n}$ from $\boldsymbol{R}^{(n)}$ such that for any circuit $C \in \boldsymbol{C}^{(n)}$, and $i \in_R \{1, \ldots, 2^n\}$, $\text{Prob}(C(i) \in S_i) < 2^{-n/4}$. Denoting this collection of 2^n sets by $\boldsymbol{S}^{(n)}$, we get that the resultant sequence $\boldsymbol{S}^{(1)}$, $\boldsymbol{S}^{(2)}, \ldots$ by a P/poly-evasive ensemble that satisfies the conditions of Theorem 3.2. We stress that considering only deterministic circuits does not restrict the generality, since we can wire in such a circuit a sequence of "random coins" that maximizes the probability $\text{Prob}(C(i) \in S_i)$.

We turn to show the existence of a collection of sets as described above. We do it by proving that there exists a positive probability to randomly choose 2^n sets $S_1, \ldots, S_{2^n}$ from $\boldsymbol{R}^{(n)}$ with the above evasivity property.

For a fixed $C \in \boldsymbol{C}^{(n)}$ and a fixed $i, 1 \leq i \leq 2^n$, consider the probability, denoted $\mathrm{Prob}_S(C(i) \in S)$, that the element $C(i)$ belongs to the set S, for S uniformly chosen over all subsets of $\{0,1\}^{4n}$ of size 2^n. Clearly,

$$\mathrm{Prob}(C(i) \in S) = 1 - \frac{\binom{2^{4n}-1}{2^n}}{\binom{2^{4n}}{2^n}} = \frac{2^n}{2^{4n}} < \frac{1}{2^{2n}}.$$

We call a set $S \subseteq \{0,1\}^{4n}$, $|S| = 2^n$, *C-bad* if there exists some $i, 1 \leq i \leq 2^n$, such that $C(i) \in S$. Fixing a circuit C, we have that for S uniformly chosen over all subsets of $\{0,1\}^{4n}$ of size 2^n,

$$\mathrm{Prob}_S(S \text{ is } C\text{-bad}) \leq \sum_{i=1}^{2^n} \mathrm{Prob}_S(C(i) \in S) < 2^n 2^{-2n} = 2^{-n}.$$

In [GK] it is proven that the measure of $\boldsymbol{R}^{(n)}$ (i.e., the proportion of sets S which are $(2^{n/4}, 2^{-n/4})$-pseudorandom) is at least $1 - 2^{-2^{n/4}}$. Therefore, for each circuit $C \in \boldsymbol{C}^{(n)}$ the probability, hereafter denoted as p_C, to uniformly choose from $\boldsymbol{R}^{(n)}$ a set S which is C-bad is

$$p_C = \mathrm{Prob}_S(S \text{ is } C\text{-bad} \mid S \in \boldsymbol{R}^{(n)}) < \frac{2^{-n}}{1 - 2^{-2^{n/4}}}.$$

We now proceed to compute the probability that for a fixed circuit $C \in \boldsymbol{C}^{(n)}$, a collection of 2^n randomly chosen sets from $\boldsymbol{R}^{(n)}$ contains a significant portion of C-bad sets. We define as "significant" a fraction $p_C + \delta_n$. (The quantity δ_n will be determined later.) Let ρ be a random variable assuming as its value the fraction of C-bad sets on a random sample of 2^n sets from $\boldsymbol{R}^{(n)}$. Clearly, the expected value of ρ is p_C. Using Hoeffding's inequality [Hoe] (see also [GK]) we get that

$$\mathrm{Prob}(\rho \geq p_C + \delta_n) \leq e^{-22^n \delta_n^2},$$

i.e., this quantity bounds the probability of choosing at random 2^n sets from $\boldsymbol{R}^{(n)}$ among which the fraction of C-bad sets is larger than $p_C + \delta_n$.

Recall that we are interested in choosing 2^n sets that are evasive for *all* circuits $C \in \boldsymbol{C}^{(n)}$. That is, we require that for *any* C, the number of C-bad sets among the 2^n sets we choose is negligible. In order to bound the probability that 2^n randomly selected sets *do not* satisfy this condition, we multiply the above probability, computed for a single circuit, by the total number of circuits in $\boldsymbol{C}^{(n)}$ which is at most $2^{(2^{n/4})^2} = 2^{2^{n/2}}$. Putting $\delta_n = 2^{-n/4}/\sqrt{2}$ we get

$$2^{2^{n/2}} \cdot e^{-22^n \delta_n^2} = 2^{2^{n/2}} \cdot e^{-22^n 2^{-\frac{n}{2}-1}} = 2^{2^{n/2}} \cdot e^{-2^{n/2}} < 1.$$

We conclude that there exists a positive probability that 2^n sets $S_1, \ldots, S_{2^n}$ chosen at random from $\boldsymbol{R}^{(n)}$ have the property that for any circuit $C \in \boldsymbol{C}^{(n)}$ the fraction of C-bad sets among $S_1, \ldots, S_{2^n}$ is less than $p_C + \delta_n$. Therefore, such a collection of sets does exist.

Finally, we bound, for this fixed collection $S_1, \ldots, S_{2^n}$, and for any circuit $C \in \boldsymbol{C}^{(n)}$, the probability $\mathrm{Prob}_i(C(i) \in S)i)$, for i randomly chosen from $\{1, \ldots, 2^n\}$. We have

$$\begin{aligned}\mathrm{Prob}_i(C(i) \in S_i) &= \mathrm{Prob}_i(C(i) \in S_i \mid S_i \text{ is } C\text{-bad}) \cdot \mathrm{Prob}_i(S_i \text{ is } C\text{-bad}) \\ &\quad + \mathrm{Prob}_i(C(i) \in S_i \mid S_i \text{ is not } C\text{-bad}) \cdot \mathrm{Prob}_i(S_i \text{ is not } C\text{-bad}) \\ &\leq 1 \cdot (p_C + \delta_n) + 0 < \frac{2^{-n}}{1 - 2^{-2^{n/4}}} + \frac{2^{-n/4}}{\sqrt{2}} < 2^{-n/4}.\end{aligned}$$

Therefore, we have shown for every circuit C of size $2^{n/4}$ that $\text{Prob}_i(C(i) \in S_i) < 2^{-n/4}$, thus proving the required properties of the sets $S_1, \ldots, S_{2^n}$.

Such a collection can be generated by a Turing machine by considering all possible collections $\{S_1, \ldots, S_{2^n}\}$ and checking whether they evade all the circuits in the set $\mathcal{C}^{(n)}$. □

Acknowledgments. Silvio Micali was one of the only researchers to strongly object to the "intuition" that the parallel versions of the interactive proof systems for quadratic residuosity, graph isomophism, and all languages in NP, are zero-knowledge. The results in this paper are rooted in this clear vision of Silvio. We are grateful to him for that.

REFERENCES

[Bab] L. Babai, *Trading group theory for randomness*, in Proc. 17th ACM STOC, 1985, pp. 421–429.

[BMO1] M. Bellare, S. Micali, and R. Ostrovsky, *Perfect zero knowledge in constant rounds*, in Proc. 22nd ACM STOC, 1990, pp. 482–493.

[BMO2] ———, *The (true) complexity of statistical zero knowledge*, in Proc. 22nd ACM STOC, 1990, pp. 494–502.

[B*] M. Ben-Or, O. Goldreich, S. Goldwasser, J. Hastad, J. Killian, S. Micali, and P. Rogaway, *Every thing provable is provable in ZK*, in Advances in Cryptology—Crypto '88 Proceedings, S. Goldwasser, ed., Lecture Notes in Comput. Sci. 403, Springer-Verlag, Berlin, 1989, pp. 37–56.

[BCC] G. Brassard, D. Chaum, and C. Crépau, *Minimum disclosure proofs of knowledge*, J. Comput. Systems Sci., 37 (1988), pp. 156–189.

[BCY] G. Brassard, C. Crépau, and M. Yung, *Everything in NP can be argued in perfect zero-knowledge in a bounded number of rounds*, in Proc. 16th ICALP, Stresa, Italy, 1989.

[CG] B. Chor and O. Goldreich, *On the power of two-point based sampling*, J. Complexity, 5 (1989), pp. 96–106.

[Fei] U. Feige, *Interactive proofs*, M.Sc. thesis, Weizmann Institute, 1987.

[FS1] U. Feige and A. Shamir, *Zero knowledge proofs of knowledge in two rounds*, in Advance in Cryptology—Crypto '89 Proceedings, Lecture Notes in Comput. Sci. 435, G. Brassard, ed., 1989, pp. 526–544.

[FS2] ———, *Witness indistinguishability and witness hiding protocols*, in Proc. 22nd ACM STOC, 1990, pp. 416–426.

[Gol] O. Goldreich, *A uniform-complexity treatment of encryption and zero-knowledge*, J. Cryptology, 6 (1993), pp. 21–53.

[GKa] O. Goldreich and A. Kahan, *How to construct constant-round zero-knowledge proof systems for NP*, J. Cryptology, to appear.

[GK] O. Goldreich and H. Krawczyk, *Sparse pseudorandom distributions*, Random Structures and Algorithms, 3 (1992), pp. 163–174.

[GMW1] O. Goldreich, S. Micali, and A. Wigderson, *Proofs that yield nothing but their validity or all languages in NP have zero-knowledge proofs*, J. Assoc. Comput. Mach., 38 (1991), pp. 691–729.

[GMW2] ———, *How to play any mental game or a completeness theorem for protocols with honest majority*, in Proc. 19th ACM STOC, 1987, pp. 218–229.

[GO] O. Goldreich and Y. Oren, *Definitions and properties of zero-knowledge proof systems*, J. Cryptology, 6 (1993), pp. 1–32.

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

[GMR1] S. Goldwasser, S. Micali, and C. Rackoff, *Knowledge complexity of interactive proofs*, in Proc. 17th ACM STOC, 1985, pp. 291–304.

[GMR2] ———, *The knowledge complexity of interactive proof systems*, SIAM J. Comput., 18 (1989), pp. 186–208.

[GS] S. Goldwasser and M. Sipser, *Private coins versus public coins in interactive proof systems*, in Advances in Computing Research: A Research Annual, Vol. 5 (Randomness and Computation, S. Micali, ed.), 1989, pp. 73–90.

[Hoe] W. Hoeffding, *Probability inequalities for sums of bounded random variables*, J. Amer. Statist. Assoc., 58 (1963), pp. 13–30.

[IY] R. Impagliazzo and M. Yung, *Direct minimum-knowledge computations*, in Advances in Cryptology—Crypto '87 Proceedings, C. Pomerance, ed., Lecture Notes in Comput. Sci. 293, Springer-Verlag, 1987, pp. 40–51.

[Jof] A. Joffe, *On a set of almost deterministic k-independent random variables*, Ann. Probab., 2 (1974), pp. 161–162.

[Ore] Y. OREN, *On the cunning power of cheating verifiers: Some observations about zero-knowledge proofs*, in Proc. 28th IEEE Symp. on OCS, 1987, pp. 462–471.

[Sim] D. SIMON, *Issues in the definition of zero-knowledge*, M.Sc. Thesis, University of Toronto, 1988.

[Sha] A. SHAMIR, IP = PSPACE, in Proc. 31st IEEE Symp. on FOCS, 1990, pp. 11–15.

[TW] M. TOMPA AND H. WOLL, *Random self-reducibility and zero-knowledge interactive proofs of possession of information*, in Proc. 28th IEEE Symp. on FOCS, 1987, pp. 472–482.

[WC] M. N. WEGMAN AND J. L. CARTER, *New hash functions and their use in authentication and set equality*, J. Comput. Systems Sci., 22 (1981), pp. 265–279.

[Yao] A. C. YAO, *How to generate and exchange secrets*, in Proc. 27th IEEE Symp. on FOCS, 1986, pp. 162–167.

SIAM J. COMPUT.
Vol. 25, No. 1, pp. 193–206, February 1996

008

THE ISOMORPHISM CONJECTURE HOLDS RELATIVE TO AN ORACLE*

STEPHEN FENNER[†], LANCE FORTNOW[‡], AND STUART A. KURTZ[§]

Abstract. The authors introduce symmetric perfect generic sets. These sets vary from the usual generic sets by allowing limited infinite encoding into the oracle. We then show that the Berman–Hartmanis isomorphism conjecture holds relative to any sp-generic oracle, i.e., for any symmetric perfect generic set A, all $\mathbf{NP}^A$-complete sets are polynomial-time isomorphic relative to A. Prior to this work, there were no known oracles relative to which the isomorphism conjecture held.

As part of the proof that the isomorphism conjecture holds relative to symmetric perfect generic sets, it is also shown that $\mathbf{P}^A = \mathbf{FewP}^A$ for any symmetric perfect generic A.

Key words. computational complexity, relativization, isomorphism conjecture

AMS subject classification. 68Q15

1. Introduction.

Is it possible to define a notion of genericity such that all **NP***-complete sets are p-isomorphic?*
Judy Goldsmith and Deborah Joseph [6]

We construct an oracle relative to which the Berman–Hartmanis isomorphism conjecture [1], [2] is true. This conjecture holds that any two **NP**-complete sets are isomorphic to one another by a polynomial-time computable and invertible one–one reduction. The isomorphism conjecture has been the subject of considerable research. We recommend the surveys by Joseph and Young [11] and Kurtz, Mahaney, and Royer [15].

The attempt to construct oracles relative to which the isomorphism conjecture either succeeded or failed began soon after the conjecture was made in 1976.

Success was first obtained in finding oracles relative to which the conjecture fails. In 1983, Kurtz (in an unpublished manuscript) constructed an oracle relative to which the conjecture failed. Hartmanis and Hemachandra [8] later combined Kurtz's construction with Rackoff's construction [18] of an oracle relative to which $\mathbf{P} = \mathbf{UP}$ (and thus no one-way functions exist [7]). In 1989, Kurtz, Mahaney, and Royer [16] showed that the conjecture fails relative to a random oracle; and Kurtz [12] gave an improved version of his original construction that showed that the conjecture fails relative to a Cohen generic oracle.

The attempt to construct an oracle relative to which the conjecture succeeds has proven much more difficult. Even partial successes have been viewed as important advances. In 1986, Goldsmith and Joseph [6] constructed an oracle relative to which a partially relativized version of the isomorphism conjecture holds. Namely, they constructed an oracle A such that all of the p-complete sets for $\mathbf{NP}^A$ are p^A-isomorphic.

An *m-degree* is an equivalence class of sets all many–one reducible to each other. In 1987, Kurtz, Mahaney, and Royer [13] gave a relativized version of their collapsing degree construction [14] and showed that there is an oracle A relative to which *some* m-degree in $\mathbf{NP}^A$ collapses. Finally, in 1989, Homer and Selman [9], [10] gave an oracle relative to which the complete degree for Σ_2^{P} collapsed.

We introduce a new notion of genericity, define the *symmetric perfect generic sets* (a.k.a. the *sp-generic sets*), and present the following theorem.

*Received by the editors May 3, 1993; accepted for publication (in revised form) August 2, 1994.

[†] University of Southern Maine, 96 Falmouth Street, Portland, ME 04103 (fenner@cs.usmsas.maine.edu). The research of this author was partially supported by NSF grant CCR 92-09833.

[‡] University of Chicago, 1100 E. 58th Street, Chicago, IL 60637 (fortnow@cs.uchicago.edu). The research of this author was partially supported by NSF grants CCR 90-09936 and CCR 92-53582.

[§] University of Chicago, 1100 E. 58th Street, Chicago, IL 60637 (stuart@cs.uchicago.edu).

THEOREM 1.1. *Relative to any symmetric perfect generic set A, all* **NP***-complete sets are polynomial-time isomorphic.*

We improve upon the work of Goldsmith and Joseph [6] by allowing **NP**-complete sets via relativized reductions.

After describing the mathematical background needed for this paper, in §3 we will describe sp-generic sets and give some of their properties. In §4 we show that $\mathbf{P}^A = \mathbf{FewP}^A$ for any sp-generic A, which will form a necessary part of our proof that the isomorphism conjecture holds relative to any sp-generic oracle. In §5 we will give some intuition for the proof of the isomorphism conjecture, followed by the detailed proof.

2. Mathematical preliminaries. The natural numbers are denoted by $\mathbf{N}$. The cardinality of a set X is denoted by $\|X\|$. Let $\Sigma = \{0, 1\}$.

We will use lower case Greek letters for partial functions from $\Sigma^* \to \{0, 1\}$. We say τ *extends* σ to mean that τ is equal to σ everywhere that σ is defined. We often identify a language $A \subseteq \Sigma^*$ as its characteristic function, for instance, in saying A extends σ. The everywhere-undefined function is denoted by $\emptyset$. Two functions are *compatible* if they agree everywhere both are defined. For compatible σ and τ, the smallest partial function extending both is denoted $\sigma \cup \tau$. We use $\mathrm{dom}(\tau)$ and $\mathrm{range}(\tau)$ to represent the domain and range of τ, respectively.

We say a computation path of an oracle Turing machine using τ is *defined* if $\tau(x)$ is defined for all queries x along that path. If M is an oracle nondeterministic Turing machine, we say that $M^\tau(x)$ accepts on a path p if all queries to the oracle made along p are in the domain of τ and are answered according to τ, and p ends in an accepting state.

We will sometimes need a machine to know the domain of τ as well as the values of τ on its domain. For these machines, we will define the total function $\overline{\tau} : \Sigma^* \times \{0, 1\} \to \{0, 1\}$ as follows:

$$\overline{\tau}(x, i) = \begin{cases} 1 & \text{if } x \in \mathrm{dom}(\tau) \text{ and } \tau(x) = i, \\ 0 & \text{otherwise.} \end{cases}$$

By abuse of terminology we will on occasion use the expression "$f^A(x)$" to refer to one of (i) the value $f^A(x)$, (ii) the function $x \mapsto f^A(x)$, or (iii) the computation of a particular machine computing f on input x using oracle A. We will try to make clear which interpretation of "$f^A(x)$" we mean when it cannot easily be inferred from context.

A one–one polynomial-time function f^A is *invertible relative to* A if there exists a polynomial-time function g^A such that for all $x \in \Sigma^*$, $g^A(f^A(x)) = x$. Note that g^A does not have to recognize the range of f^A.

For a function f and an oracle A, let $f^{A(-1)}(z)$ be the set of strings x such that $f^A(x) = z$.

Let $\mathbf{CNF}^A$ be a relativized version of **CNF** formulae (see [6]). We will also consider the formulae in a closed form, e.g., instead of a formula looking like $(x \vee y)$, it will look like $\exists x \exists y (x \vee y)$. This will allow us to talk about "true" and "false" formulae and make it easier to combine formulae with other expressions. Because we only talk about $\mathbf{NP}^A$-completeness, we only allow "$\exists$" as a quantifier. $\mathbf{SAT}^A$ consists of the true formulae relative to the oracle A.

Using standard encoding tricks and simple modifications of the theorems of Cook [4] and Berman and Hartmanis [2], we get that the following properties of $\mathbf{CNF}^A$ and $\mathbf{SAT}^A$ hold for all oracles A:

1. For every nondeterministic oracle Turing machine M that runs in time $O(n^i)$, there exists a polynomial-time unrelativized function f such that
   (a) f reduces $L(M^A)$ to $\mathbf{SAT}^A$, and
   (b) for all x, $|f(x)| = O(|x|^{2i})$.

2. Every formula $\varphi \in \mathbf{CNF}^A$ has a representation as a binary string. Every binary string represents a formula in $\mathbf{CNF}^A$.
3. Every formula represented by a binary string of n bits can only depend on A on strings of length shorter than n.
4. There is an unrelativized polynomial-time padding function P such that for all formulae φ and strings z,
   (a) $P(\varphi, z)$ is true if and only if φ is true,
   (b) $|P(\varphi, z)| > \max(|\varphi|, |z|)$, and
   (c) from $P(\varphi, z)$ we can in unrelativized polynomial time recover φ and z.

Berman and Hartmanis [2] observed that for any languages B and L such that B is $\mathbf{NP}^A$-complete and has such a padding function and L in $\mathbf{NP}^A$, there is a one-to-one length-increasing invertible reduction from L to B.

Let $i = \langle i_0, i_1 \rangle$ using the standard pairing function. Let $f_0, \dots,$ be an enumeration of functions where f_i simulates the deterministic oracle Turing machine with code i_0 running in time n^i. Let $M_0, \dots$ be an enumeration of nondeterministic oracle machines where M_i simulates the Turing machine with code i_1 running in time n^i.

We use **FP** to represent the class of polynomial-time computable functions.

3. Symmetric perfect generic sets. In this section, we define the specific type of generic set that we use in this paper. We will later show that the isomorphism conjecture holds to all such generics.

DEFINITION 3.1. *A sequence $\langle a_i \rangle_{i \in \mathbf{N}}$ of integers forms an* iterated-polynomial *sequence if there exists a polynomial p such that $p(n) \geq n^2$ for all n, $a_0 \geq 2$, and $a_{i+1} = p(a_i)$ for all i.*

DEFINITION 3.2. *A partial characteristic function $\tau : \Sigma^* \to \{0, 1\}$ is a* symmetric perfect forcing condition *if there is an iterated-polynomial sequence $\langle a_i \rangle_{i \in \mathbf{N}}$ such that*

$$\left(\bigcup_{i \in \mathbf{N}} \Sigma^{a_i} \right) \cap \operatorname{dom}(\tau) = \emptyset.$$

In other words, $\tau(x)$ is undefined for all x such that $|x| = a_i$ for some $i \in \mathbf{N}$. Note that $\tau(x)$ may be undefined on other x as well.

We generally refer to symmetric perfect forcing conditions as *sp-conditions*. As opposed to most types of forcing conditions, sp-conditions cannot necessarily be coded into finite objects.

The name *symmetric perfect* is intended to describe the topological structure of the conditions and to honor our intellectual debts. Topologically, we can view a symmetric perfect condition τ as a complete binary tree, the branchings of which correspond to points x at which $\tau(x)$ is undefined. The paths of a complete binary tree form a closed set without isolated points in their natural topology, i.e., they are *perfect*.

From a scholarly point of view, our *symmetric perfect* conditions are special cases of Gerald Sacks's *pointed perfect* conditions [20]. The unique contribution of Sacks was to recognize that forcing conditions need not be recursive (as they are in the standard finite extension arguments or in the recursion theoretic minimal degree construction). Rather, it is sufficient that τ be recursive in each of its members. This is his notion of *pointedness*. Our conditions are pointed because they can be conceived of as a complete binary tree which has been pruned at a coinfinite recursive set of points. This pruning is *symmetric* in that we remove either all of the left branchings at x (by setting $\tau(x) = 1$) or all of the right branchings at x (by setting $\tau(x) = 0$).

DEFINITION 3.3. *A set S of symmetric perfect forcing conditions is* dense *if for every sp-condition τ, there exists an sp-condition σ in S such that σ extends τ.*

DEFINITION 3.4. *A language A is* symmetric perfect generic (*sp-generic*) *if for every definable dense set S of sp-conditions, there is a $\sigma \in S$ extended by A.*

By *definable* we mean the set $\{\overline{\sigma} \mid \sigma \in S\}$ is a Π^1_1 class (see [19]).

The following theorem is a simple adaptation of the Baire category theorem.

THEOREM 3.1. *Every sp-condition τ is extended by an sp-generic language A.*

Proof. Let $D_1, \ldots$ be an enumeration of the definable dense sets. Let $\sigma_0 = \tau$. For every $i > 0$, let $\sigma_i = \sigma$ for some $\sigma \in D_i$ such that σ extends σ_{i-1}. For all $x \in \Sigma^*$, let $A(x) = \lim_{i \to \infty} \overline{\sigma}_i(x, 1)$. □

DEFINITION 3.5. *A proposition $P(A)$ is said to be* forced *by an sp-condition τ if $P(A)$ is true for all oracles A extending τ.*

Note that this definition is simpler but different from the usual definition of forcing on generic sets.

If $P(A)$ is a first-order proposition in A, then the set S of conditions that force $P(A)$ is definable since $\sigma \in S$ if and only if for all A extending σ, $P(A)$ holds.

We can already see the power of sp-generic sets by the following lemma.

LEMMA 3.2. *Given any sp-condition τ and any language X, there is an sp-condition σ extending τ such that σ forces $X \in \mathrm{P}^A$.*

Proof. Let $\langle a_i \rangle_{i \in \mathrm{N}}$ be the iterated-polynomial sequence such that τ is undefined on strings of length $\langle a_i \rangle_{i \in \mathrm{N}}$. For each i, let $b_i = a_{2i}$ and $d_i = a_{2i+1}$. Let $f(x) = x01^j$, where j is the smallest value such that $|x01^j| = d_i$ for some i. Clearly j is bounded by a polynomial in $|x|$, f is one–one and range$(f) \cap \mathrm{dom}(\tau) = \emptyset$. Define $\sigma(y)$ as

$$\sigma(y) = \begin{cases} \tau(y) & \text{if } y \in \mathrm{dom}(\tau), \\ 1 & \text{if } y = f(x) \text{ and } x \in X, \\ 0 & \text{if } y = f(x) \text{ and } x \notin X, \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Thus for any A extending σ, $x \in X$ if and only if $f(x) \in A$. The partial function σ is undefined on strings of length $\langle b_i \rangle_{i \in \mathrm{N}}$, so σ is an sp-condition. □

Of course, Lemma 3.2 does not imply that there is an sp-generic set G such that for every set X, X is polynomial-time Turing reducible to G. Lemma 3.2 only implies that all X such that the predicate "$X \in \mathrm{P}^A$" is first-order definable are encoded into all sp-generics.

4. P = FewP relative to sp-generics.

In this section, we will show that, relative to sp-generics, acceptance of nondeterministic machines with a small number of accepting paths can be decided in polynomial time.

THEOREM 4.1. *If A is an sp-generic oracle, then* $\mathbf{P}^A = \mathbf{FewP}^A$.

This proof will build on ideas from Blum and Impagliazzo [3], Hartmanis and Hemachandra [8], and Rackoff [18].

An immediate corollary follows.

COROLLARY 4.2. *For any sp-generic oracle A,* $\mathbf{P}^A = \mathbf{UP}^A$.

Let R_i be the requirement "Either there is some input x such that $M_i^A(x)$ has more than n^i accepting paths, or $L(M_i^A) \in \mathbf{P}^A$."

By our enumeration of Turing machines at the end of §2, if A satisfies R_i for all i, then $\mathbf{P}^A = \mathbf{FewP}^A$.

Fix i. The set of sp-conditions that force R_i is definable since R_i is a first-order proposition in A. We will show that the set of sp-conditions that force R_i is dense. Then any sp-generic A will extend a τ such that τ forces R_i. We will show that these sets are dense by showing how to extend any sp-condition τ to another condition σ such that σ forces R_i.

Let $M = M_i$ and let τ be an sp-condition. Suppose τ does not force "For all x, $M^A(x)$ has at most $|x|^i$ accepting paths." For some A extending τ and some x, we will have that

$M^A(x)$ has more than $|x|^i$ accepting paths. Let $\sigma = \tau \cup (A$ restricted to strings of length at most $|x|^i)$. Clearly σ extends τ and forces "For some x, $M^A(x)$ has more than $|x|^i$ accepting paths." To see that σ is an sp-condition, pick a c such that $a_c > |x|^i$ and let $b_j = a_{c+j}$ for all $j \in \mathbf{N}$.

For the rest of this section, we will assume τ forces "For all x, $M^A(x)$ has at most $|x|^i$ accepting paths."

By Lemma 3.2, there is an sp-condition σ extending τ such that σ forces $\mathbf{SAT}^{\overline{\tau}} \in \mathbf{P}^A$.

Suppose A extends σ. We will show that $L(M^A) \in \mathbf{P}^A$.

Consider the following algorithm for computing $M^A(x)$ using A as an oracle. The idea is the same as that used in [3]. We repeatedly look for *some* extension α of the partial oracle (not necessarily compatible with A) which makes M have the maximum possible number of accepting paths. To ensure consistency with A, we then answer all queries in the domain of α according to A.

In the algorithm below, we maintain the following invariants for all j:

- A extends γ_j,
- γ_{j+1} extends γ_j,
- $|\gamma_{j+1}| \leq |\gamma_j| + n^{i+1}$, and
- $\mathrm{dom}(\gamma_j) \cap \mathrm{dom}(\tau) = \emptyset$ (this fact is not crucial for the proof).

```
BEGIN ALGORITHM
  $\gamma_0 \leftarrow \emptyset$.
  FOR $j \leftarrow 0$ TO $|x|^{2i} - 1$ DO
     Let $n$ be the largest number for which there is an $\alpha$ extending $\gamma_j$ such that
        • $\alpha$ is compatible with $\tau$, and
        • $M^{\alpha\cup\tau}(x)$ has at least $n$ distinct accepting paths.
     Choose some $\alpha$ that satisfies these two conditions with minimal domain, meaning that
        dom($\alpha$) contains only those queries made along $n$ distinct accepting paths which are
        not in dom($\tau$). If $n = 0$, then $\alpha = \emptyset$.
     $\gamma_{j+1} \leftarrow$ ($A$ restricted to dom($\alpha$)).
     /* This trick is borrowed from [3]. It will be explained later. */
  ENDFOR
  $\gamma \leftarrow \gamma_{|x|^{2i}}$.        /* Note that A extends $\gamma$. */
  IF $M^{\gamma\cup\tau}(x)$ has an accepting path
     THEN accept
     ELSE reject.
END ALGORITHM.
```

Theorem 4.1 now follows from the following two lemmas.

LEMMA 4.3. *The above algorithm runs in polynomial time relative to* $\mathbf{SAT}^{\overline{\tau}}$ *and thus relative to* A.

Proof. We show that there is a fixed polynomial bound on both the size of γ_j and the running time of the jth iteration of the FOR loop for all $j < |x|^{2i}$. Assume, inside the jth iteration of the FOR loop, that γ_j has polynomial size. By our assumptions about the behavior of machine M on oracles extending τ, we have $0 \leq n \leq |x|^i$. For any such n, the question—given γ_j—of whether there exists an α extending γ_j compatible with τ such that $M^{\alpha\cup\tau}(x)$ has at least n accepting paths is an $\mathbf{NP}^{\overline{\tau}}$ question and hence can be answered by a single query to $\mathbf{SAT}^{\overline{\tau}}$ (such an α can always be chosen to have polynomial size: only include oracle queries not already in dom(τ) made along n distinct accepting paths). Thus n can be determined using polynomially many queries to $\mathbf{SAT}^{\overline{\tau}}$. Once n is found, a polynomial-size α causing $M^{\alpha\cup\tau}(x)$ to accept on n distinct paths can be constructed bit by bit in a straightforward way by making

$\mathbf{NP}^{\overline{\tau}}$ queries of the form "Given a sequence $\vec{v}$ of k bits, is there such an α whose first $k+1$ bits are $\vec{v}0$?" Similarly, we can construct the n paths. Once such an α is found, $\mathrm{dom}(\alpha)$ can be made to be minimal simply by eliminating any queries in $\mathrm{dom}(\alpha) - \mathrm{dom}(\gamma_i)$ not made along any accepting path of $M^{\alpha\cup\tau}(x)$; thus we can find a minimal α with at most polynomially many additional $\mathbf{NP}^{\overline{\tau}}$ queries. The size of $\mathrm{dom}(\alpha) - \mathrm{dom}(\gamma_j)$ is at most a polynomial in $|x|$ independent of j, so we can compute γ_{j+1} by asking polynomially many queries to A, and its size is the same as that of α. We thus have that for *all* $j \leq |x|^{2i}$, the size of γ_j and the running time of the jth iteration of the FOR loop are both bounded by a fixed polynomial in $|x|$, and thus the entire FOR loop runs in polynomial time, and $\gamma_{|x|^{2i}}$ has size polynomial in $|x|$.

Since after the FOR loop, γ has polynomial size, we can determine whether $M^{\gamma\cup\tau}(x)$ has an accepting path by asking one additional $\mathbf{NP}^{\overline{\tau}}$ question. Thus, the entire algorithm runs in polynomial time relative to A, which proves Lemma 4.3. □

LEMMA 4.4. *The above algorithm correctly decides $M^A(x)$.*

Proof. Suppose $M^A(x)$ has exactly k accepting paths. Let β be the partial function of minimal domain such that

- A extends β, and
- $M^{\beta\cup\tau}(x)$ has k distinct accepting paths.

Since $k \leq |x|^i$ and each path of $M^A(x)$ can make only $|x|^i$ queries, the size of $\mathrm{dom}(\beta)$ is at most $|x|^{2i}$ (if $k = 0$, then $\beta = \emptyset$).

CLAIM 4.5. *After the FOR loop, $\gamma_{|x|^{2i}} = \gamma$ extends β.*

Lemma 4.4 immediately follows from Claim 4.5 and the fact that A extends γ. Indeed, since $M^{\beta\cup\tau}(x)$ and $M^A(x)$ have the same number of accepting paths, we know that $M^{\gamma\cup\tau}(x)$ and $M^A(x)$ have the same number of accepting paths because extending a partial oracle can never decrease the number of accepting paths. Thus we accept if and only if $M^A(x)$ has at least one accepting path.

It remains only to prove Claim 4.5. This is similar to the incompatibility argument in [3]. It suffices to show that $\mathrm{dom}(\beta) \subseteq \mathrm{dom}(\gamma)$, since both β and γ are compatible with A. Suppose that for some $j < |x|^{2i}$ we have $\|\mathrm{dom}(\beta) - \mathrm{dom}(\gamma_j)\| = \ell > 0$. If the α chosen in the jth iteration of the FOR loop does not extend β, then it must be incompatible with β, otherwise the union $\beta \cup \alpha$ would cause $M^{\beta\cup\alpha\cup\tau}(x)$ to have at least one more accepting path than $M^{\alpha\cup\tau}(x)$ (the extra path is "contributed" by β). This contradicts the fact that α was chosen to allow the maximum possible number of accepting paths of $M^{\alpha\cup\tau}(x)$. Hence β and γ_{j+1} share at least one additional point in their domains, so $\|\mathrm{dom}(\beta) - \mathrm{dom}(\gamma_{j+1})\| \leq \ell - 1$. Since $\|\mathrm{dom}(\beta) - \mathrm{dom}(\gamma_0)\| \leq |x|^{2i}$, we must have $\|\mathrm{dom}(\beta) - \mathrm{dom}(\gamma)\| = 0$, which proves the claim. □

5. The isomorphism conjecture.

5.1. Intuition. In this section, we give some of the ideas of the proof that the isomorphism conjecture holds relative to sp-generic oracles. A full and complete proof is presented beginning in §5.2.

We first consider how researchers created oracles for which the isomorphism conjecture fails. Typically, they would create a hard function f^A and an oracle A such that $f^A(\mathbf{SAT}^A)$ is $\mathbf{NP}$-complete but not isomorphic to $\mathbf{SAT}^A$. One approach is to have f^A scramble $\mathbf{SAT}$ in a way that no reduction to $f^A(\mathbf{SAT}^A)$ could be invertible. Kurtz, Mahaney, and Royer used this approach to show that the isomorphism conjecture fails to a random oracle [16]. However, since we know that $\mathbf{P}^A = \mathbf{UP}^A$ for sp-generic oracles A (Corollary 4.2), any such scrambling function can be unscrambled.

When Kurtz showed that the isomorphism conjecture fails for regular generic oracles [12] and Hartmanis and Hemachandra created an oracle A relative to which the isomorphism conjecture fails while $\mathbf{P}^A = \mathbf{UP}^A \neq \mathbf{NP}^A$ [8], they had to use a different approach. They created functions f^A that work as follows: For φ a boolean formula represented as a string, define $\xi^A(\varphi)$ by

$$\xi^A(\varphi) = A(\varphi 01)A(\varphi 011)\ldots A(\varphi 01^n),$$

where n is the number of variables in φ. Let θ be a small true instance of $\mathbf{SAT}^A$ and define f^A by

$$f^A(\varphi) = \begin{cases} \theta & \text{if } \xi^A(\varphi) \text{ is a satisfying assignment of } \varphi, \\ \varphi & \text{otherwise.} \end{cases}$$

Note that for any oracle A and this kind of f^A, $f^A(\mathbf{SAT}^A)$ is $\mathbf{NP}^A$-complete. Using an A that encodes solutions to $\mathbf{SAT}^A$, Kurtz and Hartmanis and Hemachandra show that $f^A(\mathbf{SAT}^A)$ contains large gaps and, for reasons of density alone, cannot be isomorphic to $\mathbf{SAT}^A$.

In order to hint at how we prove our main result, we will describe how for sp-generic sets A, $f^A(\mathbf{SAT}^A)$ must be isomorphic to $\mathbf{SAT}^A$.

The oracles designed by Kurtz and Hartmanis and Hemachandra that prevent isomorphisms to $\mathbf{SAT}^A$ work by having $\xi^A(\varphi)$ be a satisfying assignment to φ. Since we are trying to create an oracle A such that the isomorphism conjecture holds, we will call the computation $f^A(\varphi)$ *bad* if $\xi^A(\varphi)$ is a satisfying assignment to φ, and all other computations $f^A(\varphi)$ we will call *good*. Note that if $f^A(\varphi)$ is good, then $f^A(\varphi) = \varphi$. Whether $f^A(\varphi)$ is good will, of course, depend on A.

Berman and Hartmanis [2] show that in order to have $\mathbf{SAT}^A$ isomorphic to $f^A(\mathbf{SAT}^A)$ we need only find a polynomial-time one–one length-increasing invertible function g^A that reduces $\mathbf{SAT}^A$ to $f^A(\mathbf{SAT}^A)$. Our $g^A(\varphi)$ will work as follows: Find a formula ψ such that

1. $|\psi| > |\varphi|$,
2. ψ is true relative to A iff φ is true relative to A,
3. $f^A(\psi)$ is good.

Then $g^A(\varphi) = f^A(\psi) = \psi$ is our reduction. The trick is for $g^A(\varphi)$ to find such a ψ.

We use a straightforward combinatorial argument to show that there exists an invertible polynomial-time function $h(\varphi, w)$ such that the following hold:

1. For all w, $|h(\varphi, w)| > |\varphi|$.
2. For all w and sp-generic A, $h(\varphi, w)$ is true relative to A if and only if φ is true relative to A.
3. For all sp-generic A, there exists a w such that $f^A(h(\varphi, w))$ is good.

Now all g^A has to do is find a w such that $f^A(h(\varphi, w))$ is good. We will use f^A to help g^A in this task.

Let $s(\varphi)$ be the formula that encodes the **NP** statement "φ is true and there exists a w such that $f^A(h(\varphi, w))$ is good." Clearly, for A, $|s(\varphi)| > |\varphi|$ and $s(\varphi)$ is true if and only if φ is true because there always is a w such that $f^A(h(\varphi, w))$ is good.

We now create $g^A(\varphi)$ as follows: Look at the computation of $f^A(s(\varphi))$. If $f^A(s(\varphi))$ is good, then output $f^A(s(\varphi)) = s(\varphi)$. Otherwise, $\xi^A(s(\varphi))$ is a satisfying assignment to $s(\varphi)$, and thus from $\xi^A(s(\varphi))$, we can obtain a w such that $f^A(h(\varphi, w))$ is good. The function g^A then outputs $f^A(h(\varphi, w)) = h(\varphi, w)$ for that w. Notice that g^A is not only length-increasing but also one–one and invertible.

Of course there is no a priori reason that a general reduction has to act like f^A. We will, however, force a general f^A to look similar to the f^A described above or not be a reduction.

Suppose f^A reduces $\mathbf{SAT}^A$ to $L(M^A)$, where f^A is an arbitrary deterministic function running in time n^i and M^A is a nondeterministic Turing machine also running in time n^i. Let us define $h(\varphi, w)$ to be a formula that encodes the following:

$$(\varphi \wedge \exists y \langle \varphi, w, y, 1\rangle \in A) \vee \exists y \langle \varphi, y, 0\rangle \in A,$$

where the y's are quantified over strings of length exactly $|\varphi|^i$. If we put at least one string of the form $\langle \varphi, w, y, 1\rangle$ into A and no strings of the form $\langle \varphi, y, 0\rangle$ into A, then φ is true if and only if $h(\varphi, w)$ is true.

We now need a notion of goodness for $f^A(h(\varphi, w))$ for arbitrary f^A. We would like to call $f^A(h(\varphi, w))$ good if $f^A(h(\varphi, w))$ fails to find a satisfying assignment to $h(\varphi, w)$. However such a thing could be hard to verify. We could, however, determine which queries to A are made by $f^A(h(\varphi, w))$. Thus we call $f^A(h(\varphi, w))$ good if $f^A(h(\varphi, w))$ does not query any string $\langle \varphi, w, y, 1\rangle$ such that $\langle \varphi, w, y, 1\rangle \in A$, i.e., $f^A(h(\varphi, w))$ does not find this part of a satisfying assignment to $h(\varphi, w)$. If $f^A(h(\varphi, w))$ is good, then we can alter the truth value of $h(\varphi, w)$ without affecting the value $f^A(h(\varphi, w))$.

Suppose $f^A(h(\varphi, w))$ is good and $q = |f^A(h(\varphi, w))| \leq |\varphi|$. Then $M^A(q)$ cannot ask questions of the form $\langle \varphi, w, y, 1\rangle$ or $\langle \varphi, y, 0\rangle$ because they are too long. We can prevent f^A from being a reduction by setting $h(\varphi, w)$ to true if $M^A(q)$ rejects or setting $h(\varphi, w)$ to false if $M^A(q)$ accepts.

Suppose $f^A(h(\psi, w_1)) = f^A(h(\theta, w_2))$ and neither of these computations ask questions about whether $\langle \psi, w_1, y, 1\rangle$, $\langle \psi, y, 0\rangle$, $\langle \theta, w_2, y, 1\rangle$, or $\langle \theta, y, 0\rangle$ are in A. Then we can prevent f^A from being a reduction by setting $h(\psi, w_1)$ to true and $h(\theta, w_2)$ to false.

We can combine the above techniques under the auspices of sp-generics to produce a reduction g that is length-increasing and almost one–one. Using the fact that $\mathbf{P}^A = \mathbf{FewP}^A$ for sp-generic A and applying this construction twice we can produce a one–one length-increasing reduction g. Grollmann and Selman [7] show that we get g invertible for free since $\mathbf{P}^A = \mathbf{UP}^A$ for sp-generic A.

5.2. Proof of the relativized isomorphism conjecture. In order to formally prove Theorem 1.1, we need the following technical lemma, whose proof we defer to §5.3.

LEMMA 5.1. *Let A be an sp-generic set, M^A a relativized nondeterministic polynomial-time Turing machine, and f^A a relativized polynomial-time reduction from* $\mathbf{SAT}^A$ *to $L(M^A)$. There is a polynomial-time function g^A and a polynomial $p(n)$ such that the following hold*:

1. *g^A reduces* $\mathbf{SAT}^A$ *to $L(M^A)$*;
2. *g^A is length increasing*;
3. *for all $q \in \Sigma^*$, if $\|g^{A(-1)}(q)\| > 1$, then*
   (a) $\|f^{A(-1)}(q)\| > 1$,
   (b) *q is in $L(M^A)$*,
   (c) $\|g^{A(-1)}(q)\| \leq p(|q|)$.

Proof of Theorem 1.1 (*assuming Lemma* 5.1). Let L be $\mathbf{NP}^A$-complete. There must exist a nondeterministic polynomial-time machine M and a polynomial-time function f such that $L = L(M^A)$ and f^A reduces $\mathbf{SAT}^A$ to L. Apply Lemma 5.1 and let g^A be the function that fulfills the properties of this lemma.

Let $T = \{q \mid \|g^{A(-1)}(q)\| > 1\}$. Note that T is in $\mathbf{FewP}^A$ because of 2 and 3(c), and thus T is in $\mathbf{P}^A$ since $\mathbf{P}^A = \mathbf{FewP}^A$ relative to sp-generic oracles (Theorem 4.1).

Let θ be a fixed member of $\mathbf{SAT}^A$ such as $(\exists x)x \vee \overline{x}$. Define $\hat{f}^A(\varphi)$ as follows:

$$\hat{f}^A(\varphi) = \begin{cases} g^A(\theta) & \text{if } g^A(\varphi) \in T, \\ g^A(\varphi) & \text{otherwise.} \end{cases}$$

Note that $\hat{f}^A$ is a reduction from $\mathbf{SAT}^A$ to L because of 3(b).

Apply Lemma 5.1, this time to $\hat{f}^A$, and let $\hat{g}^A$ be the resulting function. Note that the only possible q such that $\|\hat{g}^{A(-1)}(q)\| > 1$ is $q = g^A(\theta)$ because of 3(a).

Let $G^A(\varphi) = \hat{g}^A(P(\varphi, q))$, where P is the padding function for $\mathbf{SAT}^A$. Berman and Hartmanis [2] show that the claim below immediately implies that $\mathbf{SAT}^A$ is p^A-isomorphic to L. □

CLAIM 5.2. *The function G^A is a one–one length-increasing reduction from* $\mathbf{SAT}^A$ *to L whose inverse is computable in* $\mathbf{FP}^A$.

Clearly G^A is a reduction. Since $\hat{g}^A$ and P are length increasing, then G^A is length increasing. Also, G^A is one–one. Suppose $G^A(\varphi) = G^A(\psi)$; then $\hat{g}^A(P(\varphi, q)) = \hat{g}^A(P(\psi, q)) = q$, but this contradicts the fact that $\hat{g}^A$ is length increasing.

By Corollary 4.2, we know that $\mathbf{P}^A = \mathbf{UP}^A$. Grollmann and Selman [7] show that $\mathbf{P}^A = \mathbf{UP}^A$ implies that all one–one length-increasing polynomial-time functions relative to A are invertible relative to A. This proves the claim. □

5.3. Proof of Lemma 5.1. We define requirement R_i as follows: "Lemma 5.1 holds for $f^A = f_i^A$ and $M^A = M_i^A$." Note that by the definitions of f_i and M_i in §2, all pairs of reductions and machines will be covered by some R_i.

Fix i. Let S_i be the set of sp-conditions that force R_i. Since the statement of Lemma 5.1 is first-order definable in A, we have that S_i is a definable set of conditions. We need now show that S_i is dense. Then any sp-generic A will extend a τ such that τ forces R_i. We will show S_i is dense by showing how to extend any sp-condition τ to another condition σ such that σ forces R_i.

Fix i and let τ be an sp-condition and $\langle a_j \rangle_{j \in \mathbf{N}}$ be the corresponding iterated-polynomial sequence. We will create an sp-condition σ with corresponding sequence $\langle b_j \rangle_{j \in \mathbf{N}}$ that forces R_i. Let $f = f_i$ and $M = M_i$.

Suppose τ does not force "f^A reduces $\mathbf{SAT}^A$ to $L(M^A)$." For some A extending τ and some φ, we will have that either φ is true and $f^A(\varphi) \notin L(M^A)$ or φ is false and $f^A(\varphi) \in L(M^A)$. Let $m = \max(|\varphi|^i, |f^A(\varphi)|^i)$. Let $\sigma = \tau \cup$ (A restricted to strings of length at most m). Clearly, σ extends τ and forces "f^A does not reduce $\mathbf{SAT}^A$ to $L(M^A)$" and thus forces R_i. To see that σ is an sp-condition, pick a c such that $a_c > m$ and let $b_j = a_{c+j}$ for all $j \in \mathbf{N}$.

For the remainder of this proof we will assume that τ forces "f^A reduces $\mathbf{SAT}^A$ to $L(M^A)$."

Pick an e such that $a_{j+e} > a_j^{3i}$ for all j. Since $p(n) \geq n^2$ for all n by Definition 3.1, any $e > \log_2(3i)$ will suffice. Pick a c such that a_c is sufficiently large to avoid all the degenerate cases in this proof. For all j, let $b_j = a_{c+2ej}$ and $d_j = a_{c+2ej+e}$. This proof will never do any encoding on strings of length b_j, guaranteeing that σ is an sp-condition. In fact, all of the interesting coding for σ will occur for strings of length d_j. Initially, set $\sigma = \tau$ and also define $\sigma(x) = 0$ for every $x \notin \mathrm{dom}(\tau)$ such that x does not have length b_j or d_j for some j.

Let φ be an arbitrary $\mathbf{CNF}^A$ formula. Pick the smallest j such that $d_j > 4|\varphi|^i$. We define special tupling functions $\langle \varphi, y, 0 \rangle$, $\langle \varphi, w, y, 1 \rangle$, and $\langle \varphi, w, y, 2 \rangle$ where we are only interested in w and y as they range over strings of length $\lfloor d_j/4 \rfloor$. We design these tupling functions so that they have disjoint ranges over strings of length exactly d_j. Since $|\varphi|$, $|y|$, and $|w|$ are all bounded by $d_j/4$, such an encoding is not hard to achieve.

Let $h(\varphi, w)$ be the formula that encodes

$$(\varphi \wedge \exists y \langle \varphi, w, y, 1 \rangle \in A) \vee \exists y \langle \varphi, y, 0 \rangle \in A.$$

In other words, create a nondeterministic oracle Turing machine M such that M^A accepts if this expression is true and apply the relativized version of Cook's theorem, mentioned in §2. We will have $|h(\varphi, w)| = O(d_j^2)$.

Define $\mathbf{f}^A(h(\varphi, w))$ as follows: Simulate $f^A(h(\varphi, w))$. Whenever $f^A(h(\varphi, w))$ queries a string of the form $\langle \varphi', w', z, 1 \rangle$, $\mathbf{f}^A$ will query $\langle \varphi', w', z, 2 \rangle$.

We say the computation $\mathbf{f}^A(h(\varphi, w))$ is *good* if, for all z, $\mathbf{f}^A(h(\varphi, w))$ queries $\langle \varphi, w, z, 2 \rangle$ then $\langle \varphi, w, z, 2 \rangle \notin A$. Note that whether $\mathbf{f}^A(h(\varphi, w))$ is good does not depend on whether any string of the form $\langle \varphi', w', z, 1 \rangle$ is in A.

Let $r(\varphi)$ be the formula that encodes

$$\exists w[(\exists y \langle \varphi, w, y, 1 \rangle \in A) \text{ and } \mathbf{f}^A(h(\varphi, w)) \text{ is good}].$$

Let $s(\varphi)$ be the formula that encodes

$$(\varphi \wedge r(\varphi)) \vee \exists y \langle \varphi, y, 0\rangle \in A.$$

By suitable padding in Cook's theorem [4], we can construct s and h such that each is one–one and range$(h) \cap$ range$(s) = \emptyset$. Note that h, r, and s can be computed in unrelativized polynomial time.

LEMMA 5.3. *There is a way to set σ on the strings of length $\langle d_j \rangle_{j\in\mathrm{N}}$ such that the following hold:*

1. *σ forces "For every formula φ, $r(\varphi)$ is true."*
2. *For every φ and w, there is exactly one y such that $\sigma(\langle \varphi, w, y, 1\rangle) = 1$.*
3. *For all φ and y, $\sigma(\langle \varphi, y, 0\rangle) = 0$.*
4. *For all φ, y, and w, $\sigma(\langle \varphi, w, y, 1\rangle) = \sigma(\langle \varphi, w, y, 2\rangle)$.*

This is a combinatorial lemma that follows mainly because there are many more ways to set σ than there are extensions to σ that $\mathbf{f}^A(h(\varphi, w))$ could query. We will give a complete proof of Lemma 5.3 in §5.4.

Note that $\mathbf{f}^A(h(\varphi, w)) = f^A(h(\varphi, w))$ for all w and A where A extends σ. Also note that σ forces "For every φ and w, φ is true if and only if $h(\varphi, w)$ is true if and only if $s(\varphi)$ is true."

We now describe the algorithm for $g^A(\varphi)$.

```
BEGIN ALGORITHM
 (1) Simulate $f^A(s(\varphi))$ and let $S$ be the set of $w$ such that $f^A(s(\varphi))$ queried a string of the
     form $\langle \varphi, w, y, 1\rangle$.
 (2) If for some $w \in S$, $\mathbf{f}^A(h(\varphi, w))$ is good, then output $\mathbf{f}^A(h(\varphi, w))$ for the first such $w$.
 (3) Otherwise output $f^A(s(\varphi))$.
END ALGORITHM.
```

CLAIM 5.4. *For any A extending σ, the function g^A is a reduction from* $\mathbf{SAT}^A$ *to $L(M^A)$.*

Proof. The fact that g^A is a reduction now follows from the construction of g^A and the fact that τ forces f^A to be a reduction. □

We now show that σ forces g^A to fulfill conclusions (2) and (3)(a–c) of Lemma 5.1. We will show that if there exists an oracle A extending σ such that g^A fails to fulfill these conditions, then there exists an oracle B extending τ such that f^B does not reduce $\mathbf{SAT}^B$ to $L(M^B)$. This contradicts the assumption that τ forces f^B to be such a reduction.

We will create a B that disagrees with A only on strings longer than formulas involved in the assumed failure of some part of (2) or (3)(a–c) for g^A. This will guarantee that the truth values of these formulas will remain unchanged.

First, we show that the sp-condition σ forces that g^A is length-increasing, thus fulfilling condition (2) of Lemma 5.1.

Suppose by way of contradiction that $|g^A(\varphi)| \leq |\varphi|$. Let $q = g^A(\varphi)$. Note that $M^A(q)$ cannot look at any string of the form $\langle \varphi, y, 0\rangle$, $\langle \varphi, w, y, 1\rangle$, or $\langle \varphi, w, y, 2\rangle$ because they are too long.

We have two cases each with two subcases.

1. $q = \mathbf{f}^A(h(\varphi, w))$ output by the algorithm for $g^A(\varphi)$ in step (2):
   (a) $M^A(q)$ rejects. There must be some y such that $f^A(h(\varphi, w))$ did not query $\langle \varphi, y, 0\rangle$. Then with $B = A \cup \{\langle \varphi, y, 0\rangle\}$, f^B would not be a reduction.
   (b) $M^A(q)$ accepts. By the definition of the functions g^A and $\mathbf{f}^A$ and by parts (3) and (4) of Lemma 5.3, we have that $f^A(h(\varphi, w))$ queries $\langle \varphi, w, z, 1\rangle$ only if $\langle \varphi, w, z, 1\rangle \notin A$. Then with B equal to A minus all strings of the form $\langle \varphi, w, z, 1\rangle$, f^B would not be a reduction.
2. $q = f^A(s(\varphi))$ output by the algorithm for $g^A(\varphi)$ in step (3):
   (a) $M^A(q)$ rejects. There must be some y such that $f^A(s(\varphi))$ did not query $\langle \varphi, y, 0\rangle$. Then with $B = A \cup \{\langle \varphi, y, 0\rangle\}$, f^B would not be a reduction.

(b) $M^A(q)$ accepts. Let S be the set from the definition of g^A. Let B equal A minus all strings of the form $\langle\varphi, w, z, 1\rangle$ for $w \notin S$. If $r(\varphi)$ is false relative to B, then f^B is no longer a reduction since $f^B(s(\varphi)) = f^A(s(\varphi)) = q$, $s(\varphi)$ is false relative to B, and $q \in L(M^B)$.
Suppose $r(\varphi)$ is true relative to B. By the definition of $r(\varphi)$, we have that for some $w \in S$, $\mathbf{f}^B(h(\varphi, w))$ is good. Note that the computation of $\mathbf{f}^B(h(\varphi, w))$ is identical to the computation of $\mathbf{f}^A(h(\varphi, w))$. Since $\mathbf{f}^B(h(\varphi, w))$ is good, then $\mathbf{f}^A(h(\varphi, w))$ is also good, and so the algorithm for g^A would have output $\mathbf{f}^A(h(\varphi, w))$ in step (2).

Thus g^A is length increasing.

Suppose for some A extending σ and some q, $\|g^{A(-1)}(q)\| > 1$. Clearly, by the definition of g^A and the fact that h and s are both one–one with range$(h) \cap$ range$(s) = \emptyset$, $\|f^{A(-1)}(q)\| > 1$. Thus we have fulfilled condition (3)(a) of Lemma 5.1. We need to show how to fulfill conditions (3)(b) and (3)(c).

Suppose $q \notin L(M^A)$. Let ψ and η be such that $g^A(\psi) = g^A(\eta) = q$. We can assume without loss of generality that $|\psi| \le |\eta| < |q|$. There must be some y such that neither $g^A(\psi)$ nor $g^A(\eta)$ queries $\langle\eta, y, 0\rangle$. Let $B = A \cup \{\langle\eta, y, 0\rangle\}$. Suppose $g^A(\psi) = f^A(\nu)$ and $g^A(\eta) = f^A(\mu)$ for some formulas μ and ν. Then $f^B(\nu) = f^B(\mu) = q$ but ν is false and μ is true relative to B, and thus f^B is not a reduction. Thus we have fulfilled condition (3)(b) of Lemma 5.1.

We will now show how to fulfill condition (3)(c) with $p = t+2$ where t is the running time of g^A. Suppose $q \in L(M^A)$ and $\|g^{A(-1)}(q)\| \ge p(|q|) = t(|q|) + 2$. Let ψ be a minimum-length formula such that $g^A(\psi) = q$. By the pigeonhole principle, there is some formula $\eta \ne \psi$ such that $g^A(\eta) = q$ and $g^A(\psi)$ does not ask any queries of the form $\langle\eta, w, z, 1\rangle$ or $\langle\eta, w, z, 2\rangle$. Suppose $g^A(\psi) = f^A(\nu)$ for some formula ν.

Note that the value $f^A(\nu)$ and the truth value of ν cannot depend on whether strings of the form $\langle\eta, w, z, 1\rangle$ are in A. Since $g^A(\psi)$ simulates $f^A(\nu)$ and $g^A(\psi)$ does not ask any queries of the form $\langle\eta, w, z, 1\rangle$, then $f^A(\nu)$ does not ask any queries of the form $\langle\eta, w, z, 1\rangle$. The truth value of ν can only depend on whether strings of the form $\langle\psi, y, 0\rangle$, $\langle\psi, w, y, 1\rangle$, and $\langle\psi, w, y, 2\rangle$ are in A and the truth value of ψ and whether $\mathbf{f}^A(h(\psi, w))$ is good for some w. None of these depend on whether strings of the form $\langle\eta, w, z, 1\rangle$ are in A.

We have two cases.

1. $q = g^A(\eta) = \mathbf{f}^A(h(\eta, w))$ output by the algorithm for $g^A(\eta)$ in step (2). By the definition of g^A and $\mathbf{f}^A$ and by Lemma 5.3, we have that $f^A(h(\eta, w))$ queries $\langle\eta, w, z, 1\rangle$ only if $\langle\eta, w, z, 1\rangle \notin A$. Thus if we let B equal A minus all strings of the form $\langle\eta, w, z, 1\rangle$, the following four properties hold:
   (a) $h(\eta, w)$ is false relative to B,
   (b) ν is true,
   (c) $f^B(h(\eta, w)) = f^A(h(\eta, w)) = \mathbf{f}^A(h(\eta, w)) = g^A(\eta) = q$, and
   (d) $q = g^A(\psi) = f^A(\nu) = f^B(\nu)$.
   Thus f^B will not be a reduction.
2. $q = f^A(s(\eta))$ output in step (3) of the algorithm. Let S be the set from the definition of g^A. Let B equal A minus all strings of the form $\langle\eta, w, z, 1\rangle$ for $w \notin S$. Note that $f^B(s(\eta)) = f^A(s(\eta))$. Also, $r(\eta)$ is false relative to B for the same reasons as in case (2)(b) of the proof of condition (2) above, and thus $s(\eta)$ is also false relative to B. Thus
$$f^B(s(\eta)) = f^A(s(\eta)) = g^A(\eta) = q = g^A(\psi) = f^A(\nu) = f^B(\nu)$$
and ν is true relative to B, and thus f^B is not a reduction.

Thus we have fulfilled condition (3c). We have now fulfilled all of the conditions of Lemma 5.1. □

5.4. Proof of Lemma 5.3. We will use relativized Kolmogorov complexity for this proof. For an excellent background in Kolmogorov complexity, see the book by Li and Vitányi [17].

We need to find a σ extending τ that fulfills the conditions of Lemma 5.3. Note that by our construction of the d_j sequence, we have that $d_j > d_{j-1}^{9i^2}$. Fix j and let ℓ be the least integer such that $d_{j-1} \leq 4\ell^i$ and u be the greatest integer such that $d_j > 4u^i$. For $j = 0$, let $\ell = 0$. The values ℓ and u bound the lengths of formulae φ such that $\langle \varphi, y, 0\rangle$, $\langle \varphi, w, y, 1\rangle$, and $\langle \varphi, w, y, 2\rangle$ all have length d_j.

Let $z = |w| = |y| = \lfloor d_j/4 \rfloor$. Let $m = z2^z \sum_{\ell \leq e \leq u} 2^e$. Let x be a string of length m such that $K^{\overline{\tau}}(x) \geq m$, i.e., x is Kolmogorov random with respect to $\overline{\tau}$.

View x as a concatenation of strings $x_{\varphi,w}$ of length z, where φ ranges over all formulae of length between ℓ and u and w ranges over all strings of length z. Set $\sigma(\langle \varphi, w, x_{\varphi,w}, 1\rangle) = \sigma(\langle \varphi, w, x_{\varphi,w}, 2\rangle) = 1$ for all φ and w, and set σ to zero for all other strings of length d_j.

Clearly, this σ fulfills conditions (2)–(4) of Lemma 5.3. We still need to show that σ forces "For all formula φ, $r(\varphi)$ is true."

Suppose there is some oracle A extending σ such that for some formula φ, $r(\varphi)$ is false relative to A. We will show how to describe x with a string of length much shorter than x, contradicting the fact that x is Kolmogorov random.

Recall that $h(\varphi, w)$ has length $O(d_j^2)$. Thus $\mathbf{f}^A(h(\varphi, w))$ has running time at most $O(d_j^{2i})$. Thus $\mathbf{f}^A$ can only depend on the strings in $A' = A^{<b_{j+1}}$.

Initialize B to be A' with all the strings of the form $\langle \varphi, w, y, 2\rangle$ removed.

Create a string v as follows:

```
1. Initially set $v = \epsilon$.
2. For $w$ ranging over strings of length $z$ do the following:
   (a) For $j = 1$ to $|h(\varphi, w)|^i$,
           if the $j$th query of $\mathbf{f}^A(h(\varphi, w))$ exists and is a string $\langle \varphi, w', y, 2\rangle \in A' - B$,
           then mark $j$ and add $\langle \varphi, w', y, 2\rangle$ to $B$.
   (b) Concatenate to $v$ the number of marked $j$ followed by a list of marked $j$. Write
       these numbers with leading zeros if necessary to keep the lengths consistent in
       order to make the encoding simpler.
```

The orders in which w and j are chosen in this procedure play an important role in allowing us to keep the description of A' small.

Since $r(\varphi)$ is false relative to A then for every w, $\mathbf{f}^A(h(\varphi, w))$ queries the unique string of the form $\langle \varphi, w, y, 2\rangle$ in A (and thus in A'). Thus every such string will be added to B in step w of the above procedure if not before. At the end of this procedure, we will have $B = A'$.

The length of v is bounded by $O(2^z \log d_j)$ because there are 2^z strings in $A' - B$ initially. Note that each $\langle \varphi, w, y, 2\rangle \in A'$ can only contribute to one marking.

Now we claim that we can construct A' and thus x using an oracle for $\overline{\tau}$ and the tuple $\langle A^{\leq b_j}, v, \varphi, x'\rangle$, where x' is the concatenation of $x_{\psi,w}$ for all formulae $\psi \neq \varphi$ of length between ℓ and u, and w ranging over all strings of length z. We can reconstruct A' by repeating the procedure above using v to tell us which queries of the form $\langle \varphi, w, y, 2\rangle$ are in A'.

We can encode the tuple $\langle A^{\leq b_j}, v, \varphi, x'\rangle$ as a string of length $|A^{\leq b_j}| + |v| + |\varphi| + |x'|$ plus an additional $O(d_j)$ bits to encode the length of each piece.

The total length is bounded by $O(d_j) + 2^{b_j+1} + O(2^z \log d_j) + d_j + m - z2^z < m - O(1)$. There might exist a finite number of j such that this inequality fails. We can eliminate this possibility by an appropriate choice of a_c in the beginning of §5.3.

We have created a fixed Turing machine that outputs x with oracle $\overline{\tau}$ and input $\langle A^{\leq b_j}, v, \varphi, x'\rangle$, a tuple whose length is strictly less than $|x|$. This contradicts the fact that x was Kolmogorov random relative to $\overline{\tau}$. □

6. Conclusions and open questions. Later work by Fenner, Fortnow, Kurtz, and Li [5] show that relative to sp-generics, $\mathbf{P} = \mathbf{BPP} = \mathbf{NP} \cap \text{co-}\mathbf{NP} = \mathbf{SPP}$ but the polynomial-time hierarchy is proper. They also look at notions of genericity in a broader sense and show several interesting oracle results based on these ideas.

We have shown that relative to sp-generic oracles the isomorphism conjecture holds. Several obvious open questions remain:

- Does the isomorphism conjecture hold in the unrelativized world? Despite Theorem 1.1, the authors believe the evidence supports the position that the conjecture does not hold. Theorem 1.1 shows that a proof of this result will require nonrelatizing techniques.
- How complicated must an oracle A be such that the isomorphism conjecture holds relative to A? By Lemma 3.2 we can easily see that there are no sp-generic oracles in the arithmetic hierarchy. However, we can fulfill just the requirements necessary for the isomorphism conjecture with a set recursive in the halting problem.
- Is there a recursive oracle A? One could wonder whether we could recursively fulfill all the necessary requirements. We could use time-bounded Kolmogorov complexity in §5.4, but determining whether τ forces f to be a reduction is not decidable. However, we believe that a careful finite-injury argument could lead to a recursive oracle.
- Is there an oracle relative to which the isomorphism conjecture is true and the polynomial-time hierarchy collapses? A related question is whether there exists an oracle A such that $\mathbf{P}^A = \mathbf{UP}^A$ and $\mathbf{NP}^A = \mathbf{EXP}^A$. This oracle A also would imply that the isomorphism conjecture holds (see [10]).

Acknowledgments. This paper grew from discussions at the Dagstuhl workshop on structural complexity theory with many people, especially Ted Slaman, Bill Gasarch, Steve Homer, and Alan Selman.

We would like to thank other people for helpful discussions on the isomorphism conjecture over time: Steve Mahaney, Jim Royer, and Mike Sipser.

We also thank Richard Beigel, Bill Gasarch, Steve Homer, and Jim Royer for their comments on earlier drafts of this paper. We would also like to thank the anonymous referee for his or her extensive comments.

And, of course, we would like to acknowledge Juris Hartmanis for bringing this challenging problem to the world.

Note added in proof. John Rogers [21], extending the ideas in this paper, has created a relativized world where the isomorphism conjecture holds and $\mathbf{P} \neq \mathbf{UP}$, i.e., one-way functions exist.

REFERENCES

[1] L. Berman and J. Hartmanis, *On isomorphism and density of NP and other complete sets*, in Proc. 8th ACM Symposium on the Theory of Computing, ACM, New York, 1976, pp. 30–40.

[2] ———, *On isomorphism and density of NP and other complete sets*, SIAM J. Comput., 1 (1977), pp. 305–322.

[3] M. Blum and R. Impagliazzo, *Generic oracles and oracle classes*, in Proc. 28th IEEE Symposium on Foundations of Computer Science, IEEE, New York, 1987, pp. 118–126.

[4] S. Cook, *The complexity of theorem-proving procedures*, in Proc. 3rd ACM Symposium on the Theory of Computing, ACM, New York, 1971, pp. 151–158.

[5] S. Fenner, L. Fortnow, S. Kurtz, and L. Li, *An oracle builder's toolkit*, in Proc. 8th IEEE Structure in Complexity Theory Conference, IEEE, New York, 1993, pp. 120–131.

[6] J. Goldsmith and D. Joseph, *Three results on the polynomial isomorphism of complete sets*, in Proc. 27th IEEE Symposium on Foundations of Computer Science, IEEE, New York, 1986, pp. 390–397.

[7] J. Grollmann and A. Selman, *Complexity measures for public-key cryptosystems*, SIAM J. Comput., 17 (1988), pp. 309–355.

[8] J. Hartmanis and L. Hemachandra, *One-way functions and the nonisomorphism of NP-complete sets*, Theoret. Comput. Sci., 81 (1991), pp. 155–163.

[9] S. Homer and A. Selman, *Oracles for structural properties: The isomorphism problem and public-key cryptography*, in Proc. 4th IEEE Structure in Complexity Theory Conference, IEEE, New York, 1989, pp. 3–14.

[10] ———, *Oracles for structural properties: The isomorphism problem and public-key cryptography*, J. Comput. System Sci., 44 (1992), pp. 287–301.

[11] D. Joseph and P. Young, *Self-reducibility: Effects of internal structure on computational complexity*, in Complexity Theory Retrospective, A. Selman, ed., Springer-Verlag, Berlin, New York, 1990, pp. 82–107.

[12] S. Kurtz, *The isomorphism conjecture fails relative to a generic oracle*, Tech. report 88-018, Department of Computer Science, University of Chicago, 1988.

[13] S. Kurtz, S. Mahaney, and J. Royer, *Progress on collapsing degrees*, in Proc. 2nd IEEE Structure in Complexity Theory Conference, IEEE, New York, 1987, pp. 126–131.

[14] ———, *Collapsing degrees*, J. Comput. System Sci., 37 (1988), pp. 247–268.

[15] ———, *The structure of complete degrees*, in Complexity Theory Retrospective, A. Selman, ed., Springer-Verlag, Berlin, New York, 1990, pp. 82–107.

[16] ———, *The isomorphism conjecture fails relative to a random oracle*, J. Assoc. Comput. Mach., (1995), to appear.

[17] M. Li and P. Vitányi, *An Introduction to Kolmogorov Complexity and Its Applications*, Texts and Monographs in Computer Science, Springer, New York, 1993.

[18] C. Rackoff, *Relativized questions involving probablistic algorithms*, J. Assoc. Comput. Mach., 29 (1982), pp. 261–268.

[19] H. Rogers, *Theory of Recursive Functions and Effective Computability*, MIT Press, Cambridge, MA, 1987.

[20] G. Sacks, *Forcing with perfect closed sets*, Proc. Sympos. Pure Math., 13 (1971), pp. 331–356.

[21] J. Rogers, *The isomorphism conjecture holds and one-way funtions exist relative to an oracle*, in Proc. 10th IEEE Structure in Complexity Theory Conference, IEEE, New York, 1995, pp. 90–101.

SIAM J. COMPUT.
Vol. 25, No. 1, pp. 207–233, February 1996

009

A UNIFIED APPROACH TO DYNAMIC POINT LOCATION, RAY SHOOTING, AND SHORTEST PATHS IN PLANAR MAPS*

YI-JEN CHIANG†, FRANCO P. PREPARATA†, AND ROBERTO TAMASSIA†

Abstract. We describe a new technique for dynamically maintaining the trapezoidal decomposition of a connected planar map $\mathcal{M}$ with n vertices and apply it to the development of a unified dynamic data structure that supports point-location, ray-shooting, and shortest-path queries in $\mathcal{M}$. The space requirement is $O(n \log n)$. Point-location queries take time $O(\log n)$. Ray-shooting and shortest-path queries take time $O(\log^3 n)$ (plus $O(k)$ time if the k edges of the shortest path are reported in addition to its length). Updates consist of insertions and deletions of vertices and edges, and take $O(\log^3 n)$ time (amortized for vertex updates). This is the first polylog-time dynamic data structure for shortest-path and ray-shooting queries. It is also the first dynamic point-location data structure for connected planar maps that achieves optimal query time.

Key words. point location, ray shooting, shortest path, computational geometry, dynamic algorithm

AMS subject classifications. 68U05, 68Q25, 68P05, 68P10

1. Introduction.

A number of operations within the context of planar maps (or subdivisions, as determined by a planar graph embedded in the plane) have long been regarded as important primitives in computational geometry. First and foremost among these operations is planar point location, i.e., the identification of the map region containing a given query point; shortest-path and ray-shooting queries have also been considered very prominently.

Starting with the pioneering work in planar point location of the 1970s [10], [18], over the years several techniques have been developed, culminating in asymptotically time- and space-optimal methods [12], [17], [29] that are also of sufficiently practical flavor. Such methods, however, refer to the *static* case where no alteration of the map is allowed during its use. Due to the obvious importance of the *dynamic* setting, in recent years considerable attention has been devoted to the development of dynamic point-location algorithms [2], [6], [8], [14], [15], [21], [25], [26], [31].

All the known dynamic point-location results are for connected maps, since maintaining region names in a disconnected map would require solving half-planar range searching in a dynamic environment, for which no polylog-time algorithm is known. The best results to date for dynamic point location in an n-vertex connected map are due to Cheng and Janardan [6] and Baumgarten, Jung, and Mehlhorn [2]. The technique of [6] achieves $O(\log^2 n)$ query time, $O(\log n)$ update time, and $O(n)$ space. The data structure of [2] has query and insertion time $O(\log n \log\log n)$ and deletion time $O(\log^2 n)$, using $O(n)$ space, where the time bounds are amortized for the updates. In many real-time applications, point-location queries are executed more frequently than updates, so that it is often desirable to achieve optimal $O(\log n)$ query time in a dynamic setting. The only previous technique that supports $O(\log n)$-time queries in a dynamic environment is restricted to monotone maps [8]. For a survey of dynamic point-location techniques and other dynamic algorithms in computational geometry, see Chiang and Tamassia [9].

Algorithmic research on shortest-path and ray-shooting queries has also experienced steady progress, resulting in time-optimal techniques for the static setting [1], [5], [7], [16],

*Received by the editors January 8, 1992; accepted for publication (in revised form) August 3, 1994. An extended abstract of this paper was presented at the 4th ACM–SIAM Symposium on Discrete Algorithms, Austin, Texas, January 1993. This research was supported in part by National Science Foundation grants CCR-90-07851 and CCR-91-96176, U.S. Army Research Office grants DAAL03-91-G-0035 and DAAH04-93-0134, and the Office of Naval Research and the Defense Advanced Research Projects Agency under contract N00014-91-J-4052, ARPA order 8225.

†Department of Computer Science, Brown University, Providence, RI 02912-1910 ({yjc,franco,rt}@cs.brown.edu).

[19]. In particular, the linear-space data structures of Chazelle and Guibas [5] and of Guibas and Hershberger [16] support in $O(\log n)$ time ray-shooting and shortest-path queries, respectively, in a simple polygon with n vertices. No polylog-time method was previously known in a dynamic setting, although a polylog-time ray-shooting technique by Reif and Sen [28], designed for monotone polygons, may be extensible to the general case. Sublinear-time techniques are known only for ray-shooting queries [1], [7], with $O(\sqrt{n}\,\text{polylog}(n))$ query/update time; they support ray shooting in a set of possibly intersecting segments without taking advantage of the structure of planar maps.

A property that appears to greatly facilitate the development of dynamic point-location techniques is *monotonicity* [8], [15], [25]. Whereas the restriction to monotone maps is quite adequate for many important applications, the exclusion of more general maps is a severe shortcoming. In the static case, a connected map can be reduced to monotone (or, as we say in this paper, *normalized*) by the straightforward insertion of (auxiliary) diagonals. The same approach, when attempted for the dynamic setting, could lead to onerous updates, such as when the insertion of an edge causes the removal of a very large number of normalizing diagonals. A rather complicated and only partially documented technique due to Fries [13] is reported to assure that only a logarithmic number of normalizing diagonals be involved in any update.

In this paper we combine the feature just stated with the underpinnings of the trapezoid method, whose search efficiency both in theory [4], [23] and practice [11] is well established. This leads to the adoption of horizontal normalizing diagonals, called *lids*. The method rests on three major components:

1. a *normalization structure* that transforms a connected map into a monotone one by the addition of horizontal diagonals, while guaranteeing that no more than a logarithmic number of such diagonals are affected by insertions/deletions of edges/vertices,
2. a *hull structure* that stores the convex hulls of the chains and subchains of the monotone subregions, so that ray-shooting and shortest-path queries can be efficiently performed,
3. a *location structure* that represents a recursive decomposition of the normalized map into trapezoidal regions, and supports point-location queries in optimal time.

It is important to underscore that a single tree structure—the normalization structure—provides the unifying framework for the three applications considered. In fact, this structure, while ensuring efficient updates by controlling the size of the modifications, can be naturally augmented with node-appended secondary structures to support shortest-path and ray-shooting queries. It can also be supplemented with a distinct, but tightly coupled, location structure designed for efficient point location. The main normalization structure and its two auxiliary components act in a tightly integrated fashion. Point location is crucially used in shortest-path and ray-shooting queries and in the update of the normalization structure.

The fundamental constituents of our data structures are monotone chains and trapezoids determined by edges and horizontal lines through vertices. This provides the unifying framework for the three applications mentioned earlier. Indeed, a simple augmentation of the normalization structure provides the right environment for all three queries, as we shall illustrate. It should be underscored that, although their linkings are obviously elaborate, the elementary data structures employed are particularly simple, so that not only asymptotic efficiency is established, but also practical potential is apparent.

Our main results are outlined in the following theorem.

THEOREM 1.1. *There exists a fully dynamic data structure that supports point-location, ray-shooting, and shortest-path queries in a connected planar map $\mathcal{M}$ with n vertices. The space requirement is $O(n \log n)$. Point-location queries take time $O(\log n)$. Ray-shooting and shortest-path queries take time $O(\log^3 n)$ (plus $O(k)$ time if the k edges of a shortest path* *are reported in addition to its length*). *Updates take* $O(\log^3 n)$ *time* (*amortized for vertex updates*).

As a corollary, we can also perform stabbing queries, i.e., determine the k edges of map $\mathcal{M}$ intersected by a query segment, in $O((k+1)\log^3 n)$ time.

The contributions of this work can be summarized in the following points:

- We present the first polylog-time dynamic data structure for shortest-path queries in connected planar maps. All previous data structures for shortest paths are static and take linear time for either queries or updates when used in a dynamic environment.
- We provide the first polylog-time dynamic data structure for ray-shooting queries in connected planar maps. The previous best result is $O(\sqrt{n}\,\text{polylog}(n))$ query time.
- We present the first dynamic data structure for point-location queries in connected planar maps with optimal $O(\log n)$ query time and polylog update time. The previous best result is $O(\log n \log\log n)$ query time.
- We provide the first dynamic point-location data structure that checks the validity of an edge insertion, i.e., whether the new edge does not intersect the current edges of the map. Previous dynamic point-location data structures did not have such a capability due to the lack of an efficient dynamic ray-shooting technique.

In §2 we briefly review the terminology of planar maps and the basic data structures used by our method. The mechanics of the dynamic maintenance of a normalized map are described in §3, while §§4, 5, and 6 are respectively devoted to shortest-path, point-location, and ray-shooting queries.

2. Review of background. For the geometric terminology used in this paper, see [24]. A *connected planar map* $\mathcal{M}$ is a subdivision of the plane into polygonal regions whose underlying planar graph is connected. The map is augmented with two vertical rays, one directed toward $y = +\infty$ and the other toward $y = -\infty$, respectively issuing from the vertices of $\mathcal{M}$ with maximum and minimum y-coordinates. Thus, all but two regions of $\mathcal{M}$ are bounded simple polygons. In the following, n denotes the number of vertices of the planar map $\mathcal{M}$ currently being considered. Also, we assume that no two vertices of $\mathcal{M}$ have the same y-coordinate; the degenerate cases can be handled by standard techniques and will not be discussed in this paper.

In the plane we have an orthogonal frame of reference (x, y). A polygonal chain γ is *monotone* if any horizontal line intersects it in a single point or in a single interval or not at all. A simple polygon r is *monotone* if its boundary consists of two monotone chains. A *cusp* of a polygon is a vertex v whose internal angle is greater than π and whose adjacent vertices are both strictly above (lower cusp) or strictly below (upper cusp) v. A polygon is monotone if and only if it has no cusps. A map is monotone if all its regions are monotone.

The *trapezoidal decomposition* of a connected map $\mathcal{M}$ is obtained by drawing from each vertex v of $\mathcal{M}$ two horizontal rays that either remain unbounded or terminate when they first meet edges of $\mathcal{M}$. The resulting segments are called *splitters*. It is easily verified that a region of $\mathcal{M}$ with s vertices is partitioned by the splitters into $s-1$ trapezoids (see Fig. 1). The trapezoidal decomposition of $\mathcal{M}$ is geometrically dualized by mapping each of the obtained trapezoids τ to an arbitrary point $\delta(\tau)$ in the interior of τ. Each of the splitters is mapped to an edge between images of trapezoids in the usual way. We let $\delta(\mathcal{M})$ denote the resulting dual graph, which is a forest of trees since the trapezoids of a single region $r \in \mathcal{M}$ dualize to a tree $\delta(r)$ (because r has no holes). Note that each node of $\delta(r)$ has degree at most four. Let s_i, $i = 1, 2$, denote either a splitter or an extreme vertex of region r. Then $\text{SLEEVE}(s_1, s_2)$ denotes the union of the trapezoids traversed by the shortest path within r between any point of s_1 and any point of s_2. (Note the duality between "sleeves" in region r and paths in tree $\delta(r)$.) In a notationally consistent manner, $\delta(s)$ denotes the edge of $\delta(r)$ that is the dual of splitter s.

Our data structures are based on a variety of balanced search trees. We observe that all the standard operations on balanced search trees (insertion, deletion, split, and join) can be performed by means of a logarithmic number of more basic primitives, which we call "elementary joins and splits," defined as follows:

- An *elementary join* of two binary trees T_1 and T_2 forms a new tree T by making T_1 and T_2 the left and right subtrees of a new root node.
- An *elementary split* yields the left and right subtrees T_1 and T_2 of T by removing its root.

In particular, a simple rotation can be viewed as a sequence of four elementary splits and joins.

Three special types of data structures will be used in this paper: biased binary trees [3], $BB[\alpha]$-trees [20], and dynamic trees [30].

A *biased binary tree* [3] is a binary search tree whose leaves store weighted items. Let w be the sum of all weights. We have that the depth of a leaf with weight w_i is at most $\log(w/w_i) + 2$, and each of the following update operations can be done in $O(\log w)$ time: change of the weight of an item, insertion/deletion of an item, and split/splice of two biased trees [3].

A $BB[\alpha]$-tree [20] (where α is a fixed real, with $\frac{1}{4} < \alpha \leq 1 - \frac{\sqrt{2}}{2}$) is a binary search tree and has the following important properties (among others):

- A $BB[\alpha]$-tree with n nodes has height $O(\log n)$.
- Assume that we augment a $BB[\alpha]$-tree with secondary structures stored at its nodes. Let the subtree with root μ have ℓ leaves, and let the time for updating the secondary structures after a rotation at node μ be $O(\ell \log \ell)$. Then the amortized time of an update operation in a sequence of n insertions and deletions starting from an initially empty $BB[\alpha]$-tree is $O(\log^2 n)$.

Dynamic trees [30] are designed to represent a forest of rooted trees, with each edge directed toward the root of its tree (and called an *arc*). Some important operations (among others) supported by dynamic trees include the following:

link(μ, ν). Add an arc from μ to ν, thereby making μ a child of ν in the forest. This operation assumes that μ is the root of one tree and ν is a node of another tree.

cut(μ). Delete the arc from μ to its parent, thereby dividing the tree containing μ into two trees.

evert(μ). Make μ the root of its tree by reversing the path from μ to the original root.

Each arc of the trees is classified as *solid* or *dashed*, so that each tree is partitioned into a collection of *solid paths*, connected by dashed arcs. A solid path is maintained by a data structure called a *path tree*. Using biased binary trees [3] as the standard implementation of path trees, each of the above operations takes $O(\log n)$ time, where n is the size of the tree(s) in the forest involved.

3. The dynamics of trapezoidal decompositions. Given a connected map $\mathcal{M}$, our objective is first to systematically transform (*normalize*) it into a monotone map, and then to illustrate how to efficiently maintain it under a dynamic regimen of edge and vertex insertions/deletions.

3.1. Normalization. We first address the problem of normalization. Each region r of $\mathcal{M}$ is handled individually. We refer to a region r, bounded or unbounded. In the following, we denote by m the current number of vertices in r.

We imagine representing $\delta(r)$ as a dynamic tree $\Delta(r)$ [30] (see Fig. 1). We choose an arbitrary node of $\delta(r)$ as the *root*, which immediately forces a direction on each edge, referred to hereafter as an *arc* and directed toward the root. Since we have chosen to dualize each

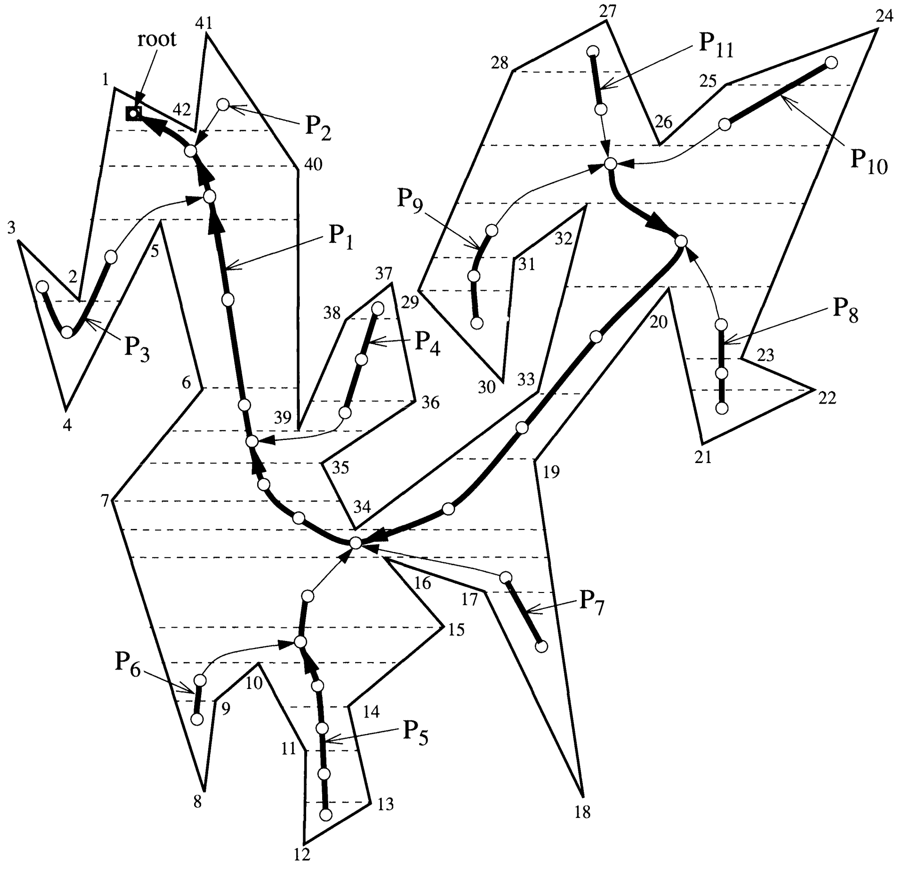


FIG. 1. *Example of a region r and its dynamic tree* $\Delta(r)$ ($P_1,\ldots,P_{11}$ *are solid paths*).

trapezoid to a point in its interior, the y-component of each arc has a well-defined sign. An arc is usually denoted either by a single letter or by an ordered pair (origin, destination). The arcs are classified as follows: letting $w(\mu)$, *weight* of μ, denote the number of nodes in the subtree rooted at node μ, an arc (μ, ν) is classified *heavy* if $w(\mu) \geq \frac{1}{2}w(\nu)$ and *light* otherwise. Consequently, at most one heavy arc enters a node of $\Delta(r)$. Note that the attributes {light, heavy} pertain uniquely to the weight structure of the dynamic tree $\Delta(r)$.

Arcs are also classified as *solid* or *dashed* to enforce the property that *at most one solid arc enters a node* of $\Delta(r)$. The maximal paths of consecutive solid arcs (possibly consisting of a single node) are called *solid paths*, and each corresponds to a sleeve of r. Note that the attributes {dashed, solid} pertain to a given, but otherwise arbitrary, decomposition of r into sleeves.

The weight structure and the sleeve decomposition are tied by the following *weight invariant*, which holds before and after the execution of data structure operations (queries or updates): heavy arcs are solid and light arcs are dashed. However, during the execution of operations, we may change heavy arcs to dashed and light arcs to solid, and thus loose the original correspondence. The weight invariant is restored at the completion of each operation.

Region r contains a set of splitters, called *lids*, which are the duals of the following arcs:

Rule 1. All dashed arcs.

Rule 2. Any two consecutive solid arcs whose y-components have opposite signs.

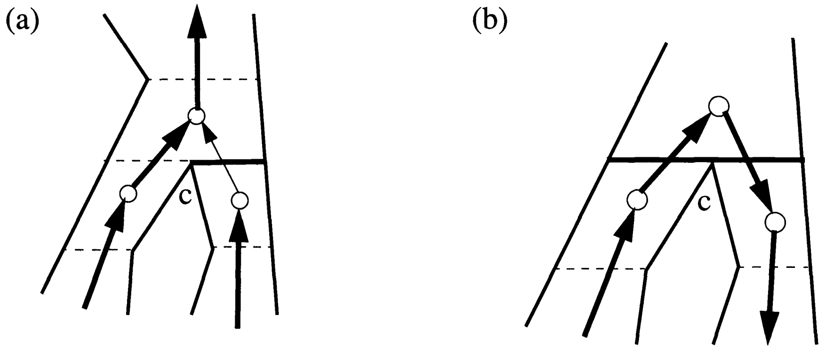


FIG. 2. *Proof of Lemma* 3.1.

Note that each lid is generated by a vertex of r. The set of lids *normalizes* r. Namely, we have the following lemma.

LEMMA 3.1. *The set of lids partitions r into a collection of monotone polygons.*

Proof. Let c be a cusp of polygon r. We consider the two arcs of $\Delta(r)$ which are the duals of the two splitters issuing from c. If at least one of them is dashed (see Fig. 2(a)), then there is at least one lid issuing from cusp c corresponding to the dashed arc (Rule 1). If on the other hand both arcs are solid, then one must have a positive y-component and the other a negative one, or otherwise they would enter the same node of $\Delta(r)$ and thus would violate the property that at most one solid arc enters a node (see Fig. 2(b)). Then these two arcs are consecutive solid arcs with y-components of opposite signs, and there are two lids from c corresponding to these arcs (Rule 2). Hence there is always at least one horizontal lid issuing from each cusp c of r, thereby achieving a decomposition of r into monotone polygons. □

LEMMA 3.2. *Each directed path of the dynamic tree* $\Delta(r)$ *contains at most* $\log_2 m$ *light arcs.*

Proof. Moving away from the root, each light arc traversed reduces the size of the current subtree by at least one half, since $w(child) < \frac{1}{2}w(parent)$. □

COROLLARY 3.3. *Any straight line drawn in region r crosses* $O(\log m)$ *lids.*

Proof. The weight invariant is always preserved before and after the execution of data structure operations. Each lid then corresponds to either (i) a light arc (Rule 1, since dashed arcs are light) or (ii) a solid arc at which the solid path containing this arc changes monotonicity with respect to the y-axis (Rule 2). By Lemma 3.2, any straight line l drawn in r crosses $O(\log m)$ lids of type (i). Now consider the lids of type (ii). Lemma 3.2 also implies that l goes through $O(\log m)$ solid paths. Observe that each solid path P can be partitioned into maximal monotone subpaths, and l can go through at most one such monotone subpath, thus crossing at most two lids of solid arcs of P. It follows that the number of lids of type (ii) crossed by l is also $O(\log m)$. □

3.2. The double-thread data structure. It is intuitively clear that insertion or deletion of an edge may substantially modify the set of trapezoids, whereas it alters only slightly the structure of region boundaries. For this reason, we adopt a data structure that represents a solid path of $\Delta(r)$ by two "threads"; these two threads respectively correspond to two chains whose union is the boundary of the sleeve associated with the solid path. The proposed structure is referred to as *double-thread data structure* for region r, denoted by $DT(r)$.

Each arc α of $\Delta(r)$ can be drawn to intersect its dual splitter issuing from some vertex v of r. Therefore we associate α with v. Notice that each vertex v in $\mathcal{M}$ is associated with two arcs: if v is a cusp of some region r, then the two splitters issuing from v both lie in r and thus cross two arcs of $\Delta(r)$; otherwise, v belongs to two regions r_1 and r_2 and the two splitters issuing from v cross respectively an arc of $\Delta(r_1)$ and an arc of $\Delta(r_2)$. Instead of maintaining the nodes of $\Delta(r)$, we choose to maintain the arcs of $\Delta(r)$ using the vertices of r as their representatives, by associating each node of $\Delta(r)$ to the arc issuing from it. As a consequence, each solid path P is represented by two binary trees *lthread*(P) and *rthread*(P), referred to as *thread trees*, whose implementation is described below. Recall that each solid path is directed toward the root. Each vertex v associated with an arc on solid path P is classified as follows: walking along P toward the root, vertex v is classified *left* if it lies to the left of P and *right* otherwise. Notice that if P is followed by a dashed arc α (every solid path except the one terminating at the root of $\Delta(r)$ has this property), then we also include α as an arc on solid path P in our representation.

The arcs of a solid path P can be partitioned into maximal monotone (on the basis of the signs of their y-components) subpaths $Q_1, Q_2, \ldots, Q_k$. Our thread trees *lthread*(P) and *rthread*(P) are each implemented as a two-level (called lower and upper) balanced binary tree (i.e., the roots of lower-level trees are leaves of the upper-level tree). Referring to *lthread*(P), in the lower level, we have a balanced binary tree *ltree*(Q_i) for each Q_i, where the leaves of *ltree*(Q_i) store the left vertices of Q_i in their path order. Thread tree *rthread*(P) is analogously organized, with *rtree*(Q_i) storing the right vertices. The roots of *ltree*(Q_i) and *rtree*(Q_i) are bidirectionally linked. In the upper level, *lthread*(P) (and analogously *rthread*(P)) has the roots of *ltree*(Q_1), *ltree*$(Q_2), \ldots,$ *ltree*(Q_k) as leaves in their path order. A bidirectional link also exists between the roots of *lthread*(P) and *rthread*(P). An example is shown in Fig. 5(a).

Any node on P might be pointed to (via dashed arcs) by some other solid paths in the dynamic tree $\Delta(r)$. Suppose that P' points to P via an arc α' associated with vertex v'. Two situations may now occur: (i) vertex v' is also associated with an arc of P (e.g., see paths P_2, P_3, and P_4 in Fig. 1 with $P = P_1$). Then v' is a left or right vertex of P (thus stored as a lower-level leaf of *lthread*(P) or *rthread*(P)). We establish a pointer from each root of *lthread*(P') and *rthread*(P') to that lower-level leaf v' (see Fig. 5(b)). The possible instances of this situation are illustrated in Fig. 3(b and d). (ii) vertex v' is not associated with an arc of P (e.g., see paths P_5 and P_7 in Fig. 1 with $P = P_1$). This occurs if P changes monotonicity (by crossing both splitters of a cusp c) at the node reached by arc α'. In this case, in order to provide a destination for the pointers from the roots of *lthread*(P') and *rthread*(P'), we introduce an auxiliary leaf, called a *coupler* (usually denoted by letter H), inserted between the two consecutive subtrees (both either *ltree*s or *rtree*s) of the thread tree not containing cusp c (see Fig. 5(b)). The possible instances of this situation are illustrated in Fig. 3(c and e).

Note that a pointer destination may be needed when a solid path P begins (Fig. 4(b and c) and Fig. 1 for $P = P_1$). In this case, we adopt the convention to insert a coupler preceding either *ltree*(Q_1) or *rtree*(Q_1), where Q_1 is the initial monotone subpath of P (see Fig. 5(b)). The overall data structure $DT(r)$ consists therefore of two rooted trees of in degree at most 4 (see Fig. 5(b)).

We now define a new parameter of nodes of $DT(r)$ (*DT-nodes*), called *charge*, which will be used to maintain the weights of the nodes of the dynamic tree $\Delta(r)$. Each DT-node corresponding to a vertex of r (a leaf of a lower-level tree) is labeled *distinguished*; the charge of a DT-node is the number of the distinguished nodes in the subtree of which it is the root.

According to its definition, the weight $w(\mu)$ of a node μ of $\Delta(r)$ is the number of the nodes in the subtree of which it is the root, or, equivalently, the number of the arcs in this subtree plus the arc α issuing from μ. It is immediate that, denoting by v the vertex associated with arc α and by Q_i the monotone subpath containing α, this number is obtained as the sum of two items: (1) the sum of the charges of all lower-level leaves (actually leaves or couplers) up to and including v in the thread tree containing v, and (2) in the other thread tree, the sum of the charges of all lower-level leaves *preceding* v^*, where v^* is the first vertex on monotone subpath Q_i whose splitter follows the splitter issuing from v, or if v^* does not exist (because

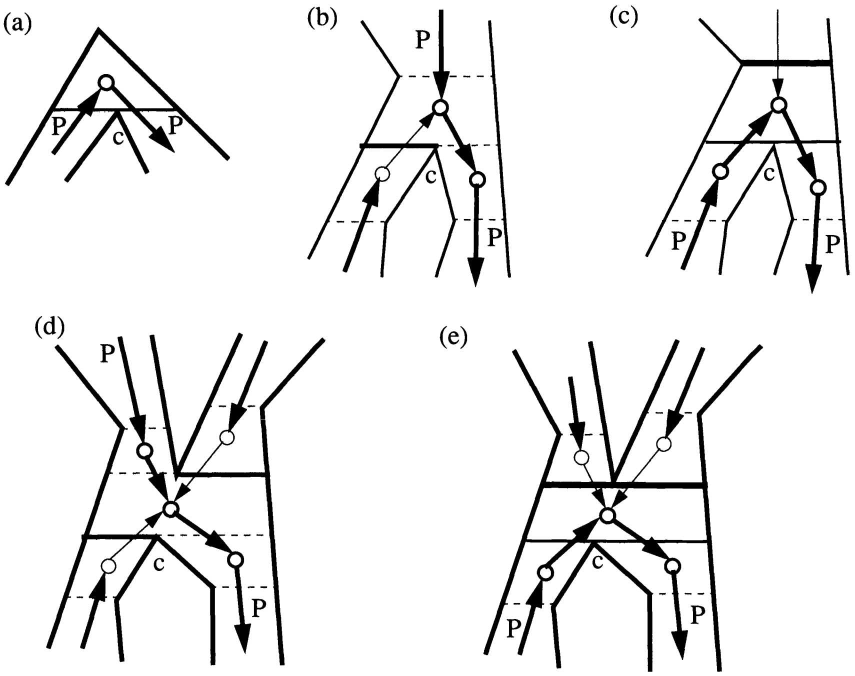


FIG. 3. *All possible cases in which a solid path P crosses a splitter issuing from a cusp c. Note that P does not change monotonicity* (*i.e., crosses only one splitter issuing from c*) *in* (b) *and* (d), *and P changes monotonicity* (*i.e., crosses both splitters issuing from c*) *in* (a), (c), *and* (e).

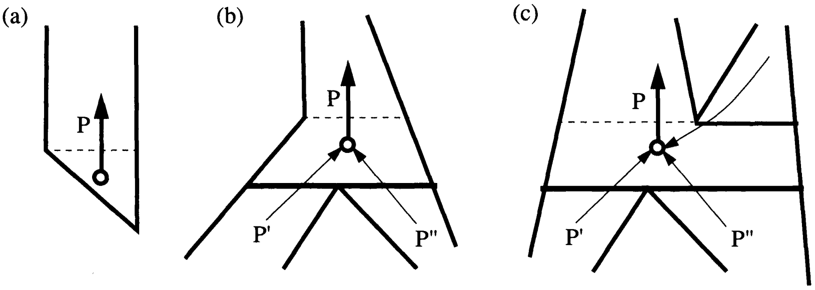


FIG. 4. *All possible cases in which a solid path P starts. Note that a coupler of P is needed to provide a destination of P' and P'' in* (b) *and* (c).

Q_i terminates at v), the sum of the charges of all lower-level leaves up to and including the last leaf of the appropriate subtree of Q_i (either $ltree(Q_i)$ or $rtree(Q_i)$). For example, let us look at $w(\mu_1)$ and $w(\mu_2)$ in Fig. 6. For $w(\mu_2)$, $v^* = s$, thus $w(\mu_2)$ is the sum of the charges of all lower-level leaves of *rthread*(P) from left up to and including v which corresponds to μ_2, and the charges of all lower-level leaves of *lthread*(P) up to and including coupler H; for $w(\mu_1)$, v^* does not exist, and thus $w(\mu_1)$ is the sum of the charges of all lower-level leaves of *rthread*(P) up to and including v which corresponds to μ_1, and the charges of all lower-level leaves of *lthread*(P) up to and including u. Clearly, we can locate v^* or decide its nonexistence in logarithmic time, using the y-coordinate of v to perform a binary search on either $ltree(Q_i)$ or $rtree(Q_i)$ of the thread tree that does not contain v.

The preceding discussion establishes the following lemma.

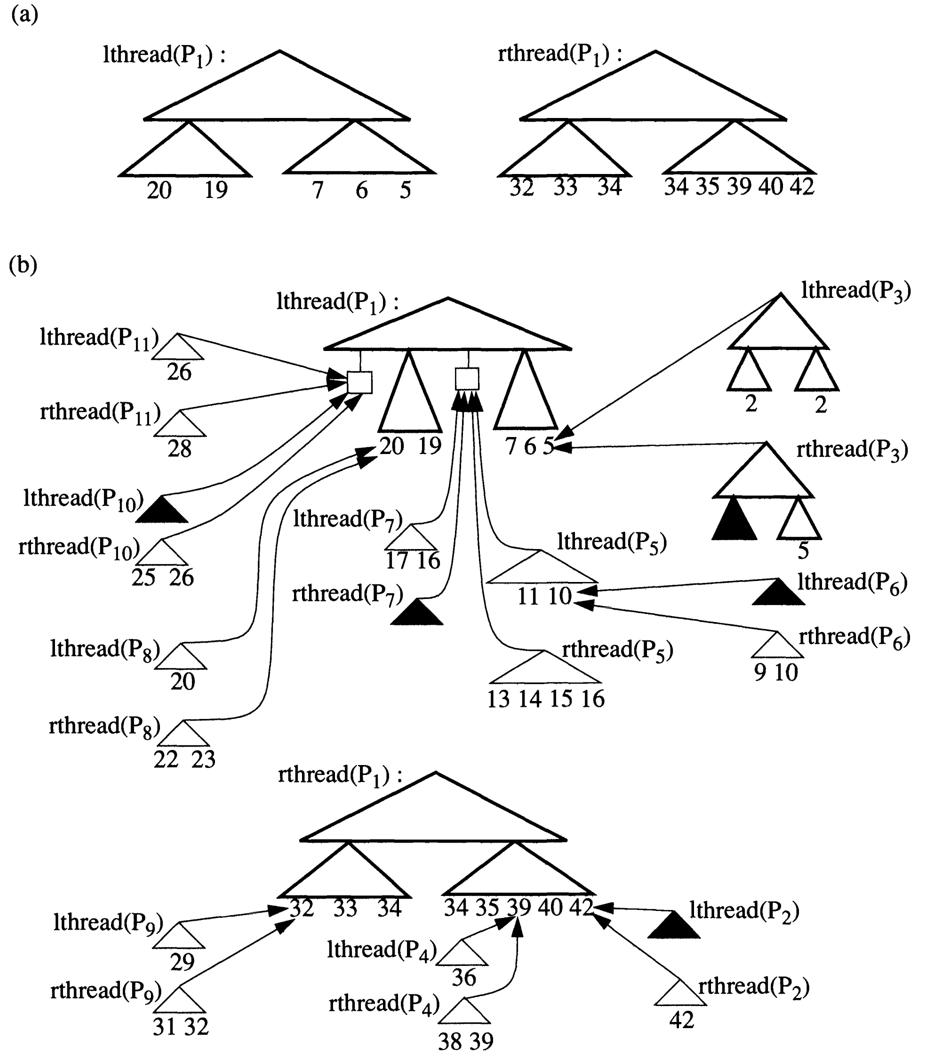


FIG. 5. *Double-thread data structure $DT(r)$ for region r in Fig.* 1: (a) *basic thread trees for* P_1; (b) *complete structure of* $DT(r)$. *The bidirectional pointers linking pairs of corresponding thread trees and thread subtrees are omitted.*

LEMMA 3.4. *The space complexity of the normalization structure for an n-vertex map is* $O(n)$.

Our data structure has an auxiliary component, called *dictionary*. The *dictionary* stores the names of vertices, edges, and regions, so that their representatives occurring in various places in the normalization structure, hull structure, location structure (see §§ 4 and 5), etc., can be efficiently accessed. The edges of a region r are also maintained in the dictionary by a balanced binary tree according to their circular order, with the root of the tree storing the name of r. We store with each edge e two pointers respectively to its left and right representatives in such trees, so that given e, the region r to its left (respectively, right) can be found by accessing its left (respectively, right) representative and walking up to the root of the tree of r. It is easy to see that accessing and updating the dictionary can be performed in logarithmic time, and

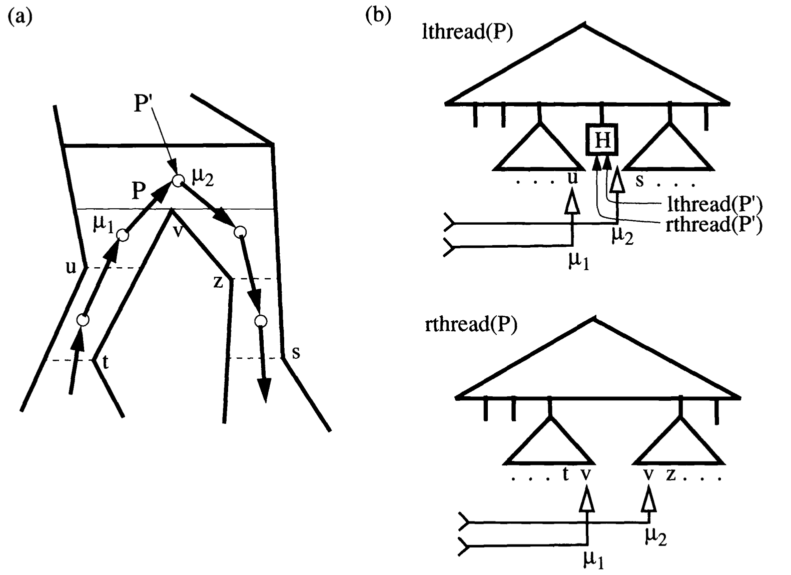


FIG. 6. *Weights $w(\mu_1)$, $w(\mu_2)$ of nodes μ_1, μ_2 of the dynamic tree $\Delta(r)$: $w(\mu_2)$ is the sum of the charges of all lower-level leaves of rthread(P) from left up to and including the occurrence of v which corresponds to μ_2, and the charges of all lower-level leaves of lthread(P) up to and including coupler H; $w(\mu_1)$ is the sum of the charges of all lower-level leaves of rthread(P) up to and including the occurrence of v which corresponds to μ_1, and the charges of all lower-level leaves of lthread(P) up to and including u.*

that the dictionary does not affect the space complexity of our data structure. Therefore we omit any further discussion of the dictionary in the rest of the paper.

3.3. Update operations. We define the following update operations on a connected map $\mathcal{M}$:

INSERTEDGE$(e, v_1, v_2, r; r_1, r_2)$. Insert edge $e = (v_1, v_2)$ into region r such that r is partitioned into two regions r_1 and r_2.

REMOVEEDGE$(e, v_1, v_2, r_1, r_2; r)$. Remove edge $e = (v_1, v_2)$ and merge the regions r_1 and r_2 formerly on the two sides of e into a new region r.

INSERTVERTEX$(v, e; e_1, e_2)$. Split the edge $e = (u, w)$ into two edges $e_1 = (u, v)$ and $e_2 = (v, w)$ by inserting vertex v along e.

REMOVEVERTEX$(v, e_1, e_2; e)$. Let v be a vertex with degree two such that its incident edges $e_1 = (u, v)$ and $e_2 = (v, w)$, are on the same straight line. Remove v and merge e_1 and e_2 into a single edge $e = (u, w)$.

ATTACHVERTEX$(v_1, e; v_2)$. Insert edge $e = (v_1, v_2)$ and degree-one vertex v_2 inside some region r, where v_1 is a vertex of r.

DETACHVERTEX(v, e). Remove a degree-one vertex v and edge e incident on v.

With the above repertory, the following theorem is immediate.

THEOREM 3.5. *An arbitrary connected map $\mathcal{M}$ with n vertices can be assembled from the empty map, and disassembled to obtain the empty map, by a sequence of $O(n)$ operations drawn from the set {point-location query,* INSERTVERTEX, REMOVEVERTEX, INSERTEDGE, REMOVEEDGE, ATTACHVERTEX, DETACHVERTEX}.

Now we show that ATTACHVERTEX and DETACHVERTEX can be simulated by a sequence of $O(1)$ operations taken among the first four of the repertory and point-location query. Referring for simplicity to ATTACHVERTEX$(v_1, e; v_2)$, we have the following emulation routine: perform

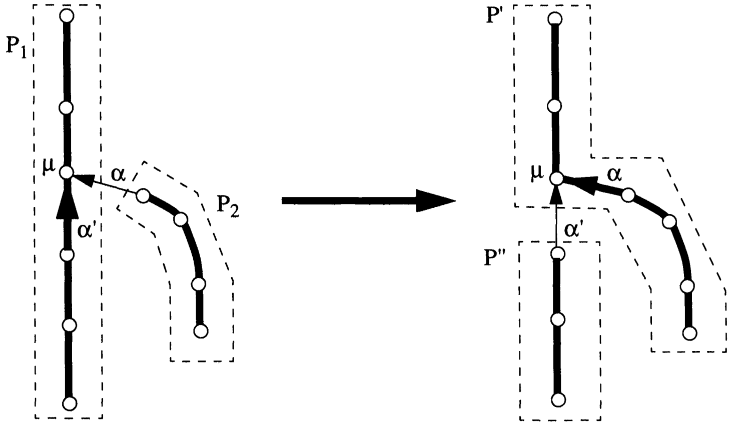


FIG. 7. *Example of splice*$(P_1, P_2; P', P'')$.

a point-location query of v_2 to obtain the region r containing it (which also provides the trapezoid containing v_2), compute the two horizontal projection points v' and v'' of v_2 on the boundary of r, insert vertices v' and v'', insert edge (v', v''), insert vertex v_2 on (v', v''), insert edge e, remove edges (v', v_2) and (v_2, v''), and finally remove vertices v' and v''.

In the rest of this section, we describe how to implement the first four operations of the above repertory on the dynamic tree $\Delta(r)$ of an arbitrary region r (a simple polygon), represented by the double-thread data structure described above.

3.3.1. Primitive dynamic-tree operations. We begin by considering some elementary dynamic-tree operations *expose*, *conceal*, and *evert* introduced in [30], in terms of which the operations of the above repertory can be immediately expressed. In the course of some updates, we may change a solid arc to dashed and vice versa and thus violate the weight invariant; thus we need the capability to restore such weight invariant. Such actions are effected by the operations *expose* and *conceal* introduced in [30]. Operation *expose*(μ), for some node μ of $\Delta(r)$, transforms the unique path P from node μ to the root of $\Delta(r)$ into a solid path, by changing the dashed arcs in P to solid and the solid arcs incident to P to dashed. Since this transformation may violate the weight invariant of dynamic trees, the inverse operation *conceal*(P) is used to remove the violation, by identifying all the light arcs in P and making them dashed, and also identifying all heavy arcs (if any) among the arcs incident to P and making them solid.

The primitive operation used in *expose* and *conceal* is *splice*$(P_1, P_2; P', P'')$, acting on two given paths P_1 and P_2 to produce two new paths P' and P'' (see Fig. 7). Originally, solid path P_2 points to node μ of solid path P_1 via a dashed arc α. Denoting by α' the (solid) arc of P_1 terminating at μ (if any), *splice* exchanges the roles of α and α', i.e., it creates two new solid paths P' and P'' with P'' pointing to P' via dashed arc α' (again, P'' and α' might be empty).

Operation *splice*$(P_1, P_2; P', P'')$ essentially involves splitting and concatenating both threads of the paths concerned. Specifically, *lthread*(P_1) is split into *lthread*(P''') and *lthread*(P''), and then *lthread*(P_2) is concatenated with *lthread*(P''') to form *lthread*(P'); this happens analogously for *rthread*. Operation *slice* may require either the insertion or the deletion of a coupler (see, for example, splicing P_4 to P_1 (insertion) and P_5 to P_1 (deletion) in Fig. 1). Since a constant number of splits/concatenations have to be performed, we have the following lemma.

LEMMA 3.6. *Operation slice can be executed in $O(\log m)$ time on the double-thread data structure.*

Since each directed path in $\Delta(r)$ contains at most $\log_2 m$ light arcs by Lemma 3.2 (each accessible by climbing to the root of an $O(\log m)$-depth thread tree), *expose* uses at most $\log_2 m$ *slice* operations and therefore is executed in $O(\log^2 m)$ time.

Given a solid path P, operation *conceal*(P) identifies the light arcs of P which have to be made dashed and the heavy arcs (if any) incident to P which have to be made solid in order to comply with the weight invariant of dynamic trees. It can be carried out by finding the topmost (i.e., closest to the root of $\Delta(r)$) light arc α, splitting P at α, removing the subpath from the root up to and including α, and then repeating the process for the remaining solid path, until no light arc is found. The heavy arcs incident to P can then be identified (and made solid) in a straightforward way: each time a light arc (μ, ν) is found, we check all (up to three) arcs incident to ν to see if any one of them is heavy; finally, we also apply this checking process to the arcs incident to the bottommost node of P. So the main issue for performing *conceal*(P) is how to find the topmost light arc.

Before describing its adaptation to the double-thread data structure, we briefly review the standard implementation of operation *conceal* as proposed by Sleator and Tarjan [30]. Let the dynamic-tree nodes of solid path P be stored left to right as the leaves of a balanced binary tree $T(P)$, called in [30] a *path tree*. Each leaf ζ of $T(P)$ stores *local_weight*(ζ), defined as the sum of the local weights of all dashed-arc children (which are the roots of some other path trees) of ζ, if any, plus 1 (to account for ζ itself). For each internal node η, *local_weight*(η) is defined as the sum of the local weights of its children. Note the similarlity between the *local weights* of nodes in path tree $T(P)$ and *charges* of nodes in thread trees *lthread*(P) and *rthread*(P) defined in §3.2. Actually, parameters *local_weight* and *charge* are identical except for their usages in computing $w(\mu)$—the weight of a dynamic-tree node μ. In $T(P)$, $w(\mu)$ is the left-to-right prefix sum of the local weights of the leaves, whereas in thread trees, $w(\mu)$ is contributed by the prefix sums of the charges of the leaves in *both* *lthread*(P) and *rthread*(P) (see §3.2). Let T_η be the subtree of $T(P)$ rooted at internal node η. Denoting by λ the rightmost leaf in T_η, and by ξ the leaf adjacent to λ on the left, variable *lefttilt*(η) is defined by *lefttilt*$(\eta) \overset{\Delta}{=} w(\xi) -$ *local_weight*(λ). We recall that arc (ξ, λ) of P is light if and only if $w(\xi) < \frac{1}{2}w(\lambda)$, i.e., $w(\xi) < \frac{1}{2}(w(\xi) +$ *local_weight*$(\lambda))$, which yields *lefttilt*$(\eta) < 0$.

Moreover, define *leftmin*$(\eta) \overset{\Delta}{=}$ *min*$\{$*lefttilt*$(\theta) : \theta$ is an internal node of $T_\eta\}$. It follows that if *leftmin*$(\eta) \geq 0$, then there is no light arc between any two adjacent leaves of T_η. Also, variable *netleft*(η) is defined as *leftmin*(η) if η is the root of $T(P)$ and *leftmin*$(\eta) -$ *leftmin*$($*parent*$(\eta))$ otherwise. Correspondingly, variables *netright*(η), *rightmin*(η), and *righttilt*(η) are defined symmetrically in a straightforward manner by summing the local weights from right to left for the purpose of reversing the path direction. In summary, each internal node η of $T(P)$ stores three values: *local_weight*(η), *netleft*(η), and *netright*(η).

To find the topmost light arc in P, we traverse a path from the root of $T(P)$ with the following advancing mechanism. Assume inductively that, for the current node η, parameter *leftmin*(η) (< 0) is known. Let η' and η'' be the left and right children of η, respectively, and ξ be the leftmost leaf of $T_{\eta''}$. From the definition

$$\textit{netleft}(\eta'') = \textit{leftmin}(\eta'') - \textit{leftmin}(\eta),$$

we obtain *leftmin*(η''). If *leftmin*$(\eta'') < 0$, then we proceed to η''. Otherwise, we compare *local_weight*(η') and *local_weight*(ξ). If *local_weight*$(\eta') <$ *local_weight*(ξ), then the arc leading to ξ is the sought light arc; else, we compute *leftmin*$(\eta') =$ *leftmin*$(\eta) +$ *netleft*(η') (which is necessarily < 0) and proceed to η' (this establishs the inductive step). By this process, akin to binary search, the topmost light arc can be found in $O(\log m)$ time. Recall that by Lemma 3.2, there are at most $\log_2 m$ light arcs in $T(P)$.

We are now ready to consider the implementation of *conceal* for the double-thread data structure. We treat thread trees *lthread*(P) and *rthread*(P) independently as two path trees, with parameter *charge* playing the role of *local weight*. By the method just illustrated, we identify at most $\log_2 m$ light arcs from each of *lthread*(P) and *rthread*(P). Note that a light arc (ξ, λ) in P assures the existence of a light arc (ξ', λ) in either *lthread*(P) or *rthread*(P) that contains leaf λ, where ξ' is the left-neighboring leaf of λ. Indeed, in $T(P)$, the sum $w(\xi)$ of the local weights up to and including ξ satisfies $w(\xi) < \mathit{local\_weight}(\lambda)$. But in the appropriate thread tree (i.e., either *lthread*(P) or *rthread*(P) that contains λ), the sum w' of the charges up to and including ξ' is only a fraction of $w(\xi)$ ($w(\xi)$ is contributed by *both* *lthread*(P) and *rthread*(P)), so that $w' \leq w(\xi) < \mathit{local\_weight}(\lambda) = \mathit{charge}(\lambda)$, and (ξ', λ) is light. Hence $2\log_2 m$ light arcs from *lthread*(P) and *rthread*(P) give all possible candidates for light arcs in P. For each such candidate (ξ', λ), we perform a binary search in the paired thread tree to locate the point just before λ, at the same time accumulate the total charge w'' up to and including this point in that tree, then compute $w(\xi)$ by adding w'' to w', and check if $w(\xi) < \mathit{charge}(\lambda)$ $(= \mathit{local\_weight}(\lambda))$. Therefore, we find $2\log_2 m$ candidates, perform $2\log_2 m$ binary searches for checking, identify at most $\log_2 m$ light arcs in P (and also at most $\log_2 m + 1$ heavy arcs incident to P), and then split and join P accordingly—each of these operations within $O(\log m)$ time. This leads to the following lemma.

LEMMA 3.7. *The update of the double-thread data structure as required by the operation conceal can be performed in* $O(\log^2 m)$ *time.*

Operation *evert*(μ), for an arbitray node μ of $\Delta(r)$, moves the root of $\Delta(r)$ to μ while preserving the weight invariant. If we can reverse the direction of a solid path, then *evert*(μ) can be carried out as follows: we perform *expose*(μ) to obtain a solid path P from μ to the original root, reverse the direction of P (which effectively moves the root to μ), and then perform *conceal*(P) to comply with the weight invariant. We add a "direction" bit to each node of the thread trees, so that when we reverse the direction of a solid path P, the direction bit of the root of *lthread*(P) is complemented, indicating that the meanings of left and right subtrees of *lthread*(P) are interchanged; this is done similarly for the direction bit of the root of *rthread*(P). Also, these two complemented bits indicate that *lthread*(P) means *rthread*(P) and vice versa. Given the direction bits and operations *expose* and *conceal*, we can perform *evert* in the double-thread data structure in $O(\log^2 m)$ time.

In the following, if μ is a node of $\Delta(r)$ and a an incoming arc of μ, the notations *expose*(a) and *expose*(μ) are equivalent, and similarly for *evert*.

LEMMA 3.8. *Given splitters* s_1 *and* s_2 *of region* r *with* m *vertices,* SLEEVE(s_1, s_2) *and the corresponding solid path between* $\delta(s_1)$ *and* $\delta(s_2)$ *can be constructed in* $O(\log^2 m)$ *time, by means of* $O(\log^2 m)$ *elementary splits/joins of thread trees.*

Proof. We obtain a solid path between $\delta(s_1)$ and $\delta(s_2)$ by *evert*$(\delta(s_1))$ and *expose*$(\delta(s_2))$. Each of operations *evert* and *expose* uses $O(\log^2 m)$ elementary splits/joins and takes $O(\log^2 m)$ time. □

The double-thread structure adds two new primitive operations to the original repertory of dynamic trees. Operation *part*$(P, e;\ P_1, P_2)$ on a solid monotone path P separates *lthread*(P) and *rthread*(P), and creates two new solid paths P_1 and P_2 by adjoining *lthread*(P) and *rthread*(P) to a new edge e. Namely, *lthread*(P_1) = *lthread*(P), *rthread*$(P_1) = e$, *lthread*$(P_2) = e$, and *rthread*(P_2) = *rthread*(P). The operation *pair*$(P_1, P_2;\ P, e)$ is the inverse operation of *part*$(P, e;\ P_1, P_2)$ and is implemented similarly.

LEMMA 3.9. *Operations part and pair have time complexity* $O(1)$.

As we shall see in the next section, operations *part* and *pair* are crucial in the efficient execution of INSERTEDGE and REMOVEEDGE.

3.3.2. Insertion and deletion of edges and vertices. Operation INSERTEDGE(e, v_1, v_2, r; r_1, r_2) is carried out as follows:

1. For $i = 1, 2$, if v_i is an extreme vertex of r, let $s_i = v_i$, else let s_i be a splitter of r induced by v_i. If v_i is a cusp of r, then there are two such splitters; by viewing edge $e = (v_1, v_2)$ as issuing from v_i, s_i is taken as the left splitter of v_i if e goes toward left (and as the right splitter otherwise), so that SLEEVE(s_1, s_2) is the smallest monotone sleeve that contains e.
2. Construct SLEEVE(s_1, s_2) and the corresponding solid path P by performing $evert(\delta(s_1))$ and then $expose(\delta(s_2))$.
3. Insert edge e by performing $part(P, e; P_1, P_2)$, so that there are new solid paths P_1 and P_2 respectively in new regions r_1 and r_2.
4. For each of the (up to three) solid paths previously pointing to the head of P, make it point to the head of P_1 if it lies in r_1, and to the head of P_2 if it lies in r_2; do this similarly for the solid paths previously pointing to the tail of P. Note that P_1 and P_2 have the same orientation as P.
5. Create a new dynamic tree $\Delta(r_1)$ for r_1, by putting the root at the end of P_1 that is closer to v_1 (which does not change the direction of P_1), then performing operation $conceal(P_1)$; similarly create a new dynamic tree $\Delta(r_2)$. Note that the *conceal* operations readily splice the solid paths pointing to the heads and tails of P_1 and of P_2 if necessary.

We analyze the time complexity of the above operation. Steps 1, 3, and 4 take $O(1)$ time, and the other steps globally involve a fixed number of *evert*, *expose*, and *conceal* operations, so that the total time required for updating the double-thread data structure is $O(\log^2 m)$.

Operation REMOVEEDGE(e, v_1, v_2, r_1, r_2; r) is the inverse operation of INSERTEDGE. We first evert v_1 and then expose v_2 in both $\Delta(r_1)$ and $\Delta(r_2)$, pair the two solid paths into one, and conceal it. This can also be done in $O(\log^2 m)$ time.

Operation INSERTVERTEX(v, e; e_1, e_2) is performed as follows. We insert v with $charge(v) = 1$ into $lthread(P)$ and $rthread(P')$ for some solid paths P and P' of different regions r and r', where both $lthread(P)$ and $rthread(P')$ contain two endpoints v_1 and v_2 of e. In dynamic tree $\Delta(r)$, we perform *expose* on the one of v_1 and v_2 that is farther from the root to obtain a solid path, and then perform *conceal* on this path; in $\Delta(r')$ we perform exactly the same operations. It is easy to see that operation INSERTVERTEX is executed in $O(\log^2 m)$ time. Operation REMOVEVERTEX is the inverse operation of INSERTVERTEX and can be completed within the same time bound.

4. Shortest-path queries. In this section, we illustrate how the normalization data structure can be modified, by appending secondary data structures collectively called *hull structure*, to answer the following queries:

PATHLENGTH(q_1, q_2, r). Return the length of a shortest path inside region r between query points q_1 and q_2.

PATH(q_1, q_2, r). Return the shortest path inside region r between query points q_1 and q_2 as a chain of segments.

First, by point location (see §5) we can check whether q_1 and q_2 belong to r. Note that we need to specify within which region the shortest path is sought to avoid ambiguities when both q_1 and q_2 belong to edges of the subdivision. We now show that the above queries can be supported in worst-case time $O(\log^3 n)$ and $O(\log^3 n + k)$, respectively, where k is the number of segments in the shortest path reported by PATH.

The notion of *hourglass* is central to our current problem. We adopt the terminology proposed by Guibas and Hershberger [16].

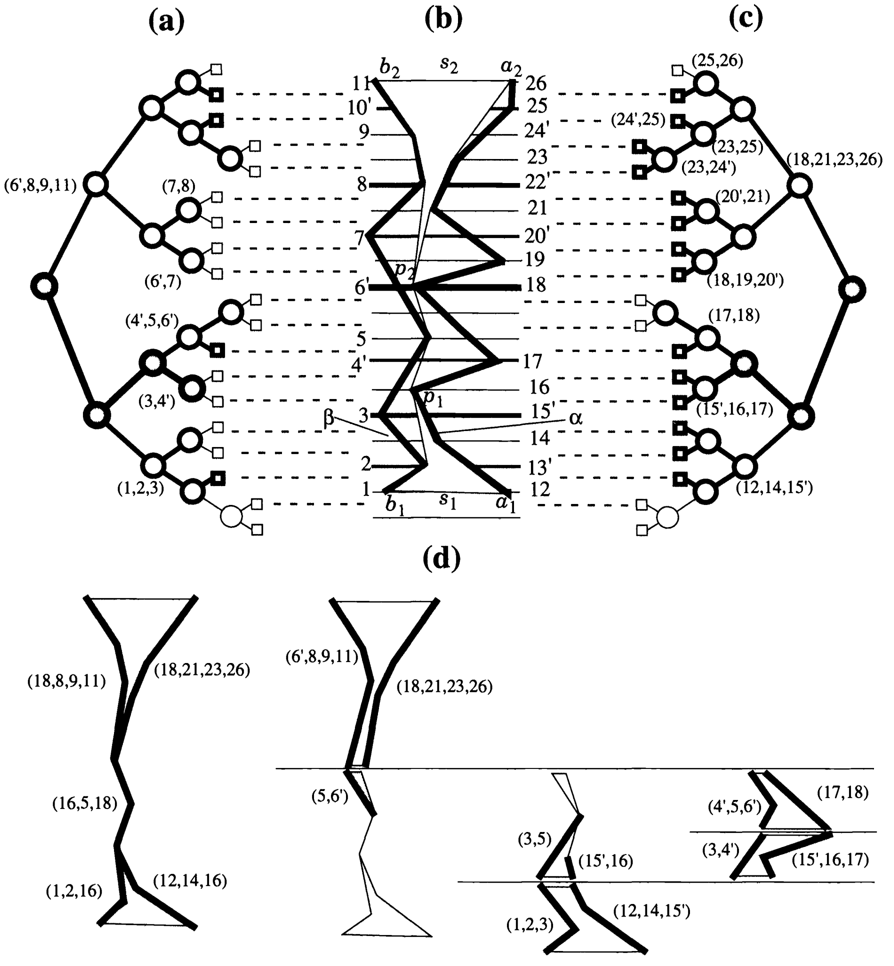


FIG. 8. *Example of representation of hourglasses in the nodes of ltree*(Q) *and rtree*(Q) *of a monotone path* Q. (b) *The sleeve of* Q (*directed from left to right*)*: the parallel lines drawn on it represent set* Y; *the points on the sleeve with labels of the type* i' *delimit fragments of the same edge*; *the hourglass between the extreme splitters of the sleeve is shown grey filled.* (a) *Pruned tree ltree*(Q)*: the nodes of ltree*(Q) *are those drawn with thick lines, while the nodes drawn with thin lines denote the subtrees of* $\mathcal{Y}$ *pruned away to construct ltree*(Q); *the grey-filled nodes are associated with closed sleeves, and the white-filled nodes are associated with open sleeves. Next to each white-filled node* μ *we show the subchain of* HOURGLASS(μ) *stored at* μ. (c) *Pruned tree rtree*(Q) (*similar comments as in* (a) *apply*). (d) *Hourglasses of the grey-filled nodes and of their children. The subchains stored at each node are labeled and shown with thick lines.*

Consider two nonintersecting diagonals $s_1 = (a_1, b_1)$ and $s_2 = (a_2, b_2)$ of r, where the endpoints have been named so that the counterclockwise cyclic sequence of points in the boundary of r includes the subsequence (a_1, a_2, b_2, b_1). The *hourglass* of s_1 and s_2, denoted HOURGLASS(s_1, s_2), is the subregion of r formed by the union of all the shortest paths PATH(q_1, q_2, r) with $q_1 \in s_1$ and $q_2 \in s_2$ (see Fig. 8(b)). It is known that the boundary of HOURGLASS(s_1, s_2) is the concatenation of s_1, PATH(a_1, a_2, r), s_2, and PATH(b_2, b_1, r). Let α be the subchain of r counterclockwise from a_1 to a_2, and define β similarly for b_2 and b_1. The hourglass has one of the following special structures (as analyzed in [16]):

Open hourglass. If the convex hulls inside r of α and β do not intersect, then PATH(a_1, a_2, r) is the convex hull of the subchain of α clockwise from a_1 to a_2, and similarly for PATH(b_2, b_1, r).

Closed hourglass. If the convex hulls of α and β intersect, then there exist vertices p_1 and p_2 of $\alpha \cup \beta$ such that $\text{PATH}(a_1, a_2, r) \cap \text{PATH}(b_1, b_2, r) = \text{PATH}(p_1, p_2, r)$. Without loss of generality, assume that p_1 is in α. Then $\text{PATH}(a_1, p_1, r)$ is the convex hull inside r of the subchain of α from a_1 to p_1, while $\text{PATH}(b_1, p_1, r)$ is the union of segment (p_1, p_1') and the convex hull inside r of the subchain of β from b_1 to p_1', where p_1' is the vertex of β closer to b_1 on the two tangents from p_1 to β. Similar arguments apply to p_2. The union of $\text{PATH}(a_i, p_i, r)$ and $\text{PATH}(b_i, p_i, r)$ $(i = 1 \text{ or } 2)$ is called a *funnel* [19]. Vertices p_1 and p_2 are called the *apices* of the hourglass, and the path between them the *string* of the hourglass (see Fig. 8(b)).

If we represent an hourglass by its string and the (two to four) convex chains forming the rest of its boundary, and for each polygonal chain represented, we also store its length, then given $\text{HOURGLASS}(s_1, s_2)$, it is possible to compute $\text{PATHLENGTH}(q_1, q_2, r)$ in $O(\log n)$ time for any two points $q_1 \in s_1$ and $q_2 \in s_2$ by means of $O(1)$ common-tangent computations. Also, given $\text{HOURGLASS}(s_1, s_2)$ and $\text{HOURGLASS}(s_2, s_3)$ in r, with s_1 and s_3 on opposite sides of the line containing s_2, it is possible to compute $\text{HOURGLASS}(s_1, s_3)$ in time $O(\log n)$ by means of $O(1)$ common-tangent computations and $O(1)$ split and join operations on the chains forming the two hourglasses.

We now consider the modifications of the normalization data structure that enable the support of the given path queries. As we shall see, only three items are needed, i.e.,

(i) the choice of an appropriate implementation of the trees *ltree* and *rtree* introduced in §3.2;

(ii) the appending of secondary data structures (collectively called "hull structure") to the nodes of *ltree*s and *rtree*s. The *hull structure* stores at the nodes of *ltree*s and *rtree*s the hourglasses of the corresponding sleeves; it establishes an implicit correspondence between the two chains of a monotone sleeve, allowing both efficient access to the hourglass of the sleeve and fast pairing or parting of the two chains as required by edge insertion or deletion;

(iii) a separate BB[α]-tree $\mathcal{Y}$ (called *Y-tree*) that determines a hierarchical partition of the plane into horizontal strips, according to which *ltree*s and *rtree*s are implemented.

We first describe the adopted representation of polygonal chains. A concatenable queue, called *chain tree*, will be used to represent a polygonal chain γ. The chain tree T for γ is a balanced tree and has in-order thread pointers. Each node μ of T corresponds to a subchain γ_μ of γ and stores the endpoints of γ_μ, the common point of the subchains of the children of γ_μ, and the length of γ_μ. It should be clear that this information can be updated in $O(1)$ time per elementary join or split, so that splitting or splicing two chain trees takes logarithmic time. With this representation, it is possible to find the two tangents from a point to a convex chain and the four common tangents between two convex chains in logarithmic time [24].

We now give the details of our representation of hourglasses. An open hourglass is represented by storing its two convex chains into chain trees. A closed hourglass is represented by storing into separate substructures the four convex chains forming the funnels, and the string between the apices. The convex chains of the funnels and the string are each stored into a chain tree.

Without loss of generality, we assume that the degree of each vertex of $\mathcal{M}$ is at most 3. This is not restrictive since we can expand a vertex v with degree $d > 3$ into a chain of degree-3 vertices connected by edges of infinitesimal length. Since the sum of the degrees of all vertices of $\mathcal{M}$ is $O(n)$, the total number of vertices after the expansion is still $O(n)$. Every update operation in the original map $\mathcal{M}$ can be simulated with $O(1)$ operations in the modified map with bounded-degree vertices.

We consider the ordered sequence Y of the y-coordinates of the vertices of $\mathcal{M}$ and establish a one-to-one correspondence between Y and the leaves of a BB[α]-tree $\mathcal{Y}$, called *Y-tree*, which is added as a separate tree into the data structure. Tree $\mathcal{Y}$ determines a hierarchical partition of the plane into horizontal strips according to the well-known segment-tree scheme. Each node of $\mathcal{Y}$ corresponds to a *canonical interval* of y-coordinates. A vertical interval (y', y'') with $y', y'' \in Y$ is uniquely partitioned into $O(\log n)$ canonical intervals, called the *fragments* of (y', y''), and their associated nodes in $\mathcal{Y}$ are called the *allocation nodes* of (y', y''). We extend this terminology to any geometric entity that is uniquely associated with a vertical interval, such as an edge, a monotone chain, or a monotone sleeve.

We now introduce the useful notion of the "pruned tree." A *pruned tree* of a rooted tree T is a tree S that can be obtained from T by removing from it the subtrees rooted at a selected subset of its nodes. Pruned trees of a balanced tree T support the full repertory of concatenable queue operations. Each operation takes $O(\log n)$ time and is performed by means of $O(\log n)$ elementary joins and splits between pruned trees whose roots are associated with sibling nodes in T. A sequence I of k consecutive intervals with endpoints in Y will be stored in a pruned subtree of $\mathcal{Y}$, whose leaves are the allocation nodes of the intervals of I and whose internal nodes are the ancestors of such leaves. It is easy to verify that the pruned tree associated with I has $O(k \log n)$ nodes and $O(\log n)$ height.

Now, we show how to modify the normalization structure so that hourglasses can be dynamically maintained (see Fig. 8). We denote with Q a maximal monotone subpath of a solid path P and specify the implementation of *ltree*(Q) and *rtree*(Q). We use pruned trees augmented with chain trees as secondary structures. Our scheme uses ideas from [22] and [16].

- Trees *ltree*(Q) and *rtree*(Q) are implemented by means of pruned trees with respect to $\mathcal{Y}$.
- Let μ be a node of *ltree*(Q) (nodes of *rtree*(Q) are handled identically) and ν the parent of μ. Node μ has a pointer to the corresponding node y of $\mathcal{Y}$. Also, if μ is not a leaf, then we establish a back pointer from y to μ. We do not set up back pointers from y to leaves of *ltree*(Q) (or of *rtree*(Q)) in order to obtain efficient updates, as we shall see later. Consider the subpath Q' of Q associated with the subtree of *ltree*(Q) rooted at μ. We denote with SLEEVE(μ) the sleeve of Q', with s_1 and s_2 the splitters that delimit SLEEVE(μ), with HOURGLASS(μ) the hourglass of s_1 and s_2 (namely, HOURGLASS(s_1, s_2)), and with CHAIN(μ) the "left chain" of SLEEVE(μ), i.e., the chain formed by the edge fragments stored at the leaves of the subtree of *ltree*(Q) rooted at μ.

  We distinguish several subcases:
  - If μ is a leaf of *ltree*(Q), then μ stores the corresponding edge fragment.
  - If HOURGLASS(μ) is open and HOURGLASS(ν) is closed, then μ stores in a secondary data structure the right convex hull of CHAIN(μ).
  - If both HOURGLASS(μ) and HOURGLASS(ν) are open, then μ stores in a secondary data structure only the endpoints of CHAIN(μ) and the portion of the right hull of CHAIN(μ) that is not stored at an ancestor of μ.
  - If HOURGLASS(μ) is closed and μ is the root of *ltree*(Q), then μ stores in secondary data structure the (up to five) components of HOURGLASS(μ).
  - If HOURGLASS(μ) is closed and μ is not the root, then μ stores the apices and the length of the string of HOURGLASS(μ), plus the subchains of the funnels of HOURGLASS(μ) that are not stored at the ancestors of μ.
- The upper levels (see §3.2) of thread trees *lthread*(P) and *rthread*(P) are essentially identical (except for the couplers). Also, an internal node μ in the upper level of *lthread*(P) stores the length and the endpoints of the string of HOURGLASS(μ). The corresponding node of *rthread*(P) stores exactly the same information.

LEMMA 4.1. *The space requirement of the hull structure is* $O(n \log n)$.

Proof. We only need to determine the space used by the secondary structures (the chain trees) that augment the *ltree*s and *rtree*s. Consider the set S of all segments s such that s is either an edge fragment or the tangent segment in the hourglass of a node in *ltree* or *rtree*. We claim that the size of S is $O(n \log n)$. By standard segment-tree arguments, the number of edge fragments in S is $O(n \log n)$. For the tangent segments, consider the hourglasses HOURGLASS(μ), HOURGLASS(μ'), and HOURGLASS(μ'') of a node μ and its children μ' and μ''. Note that SLEEVE(μ') and SLEEVE(μ'') share a common splitter, say s_2, and the other splitters s_1 of SLEEVE(μ') and s_3 of SLEEVE(μ'') lie on opposite sides of s_2. It follows that HOURGLASS(μ) = HOURGLASS(s_1, s_3) is obtained from HOURGLASS(μ') = HOURGLASS(s_1, s_2) and HOURGLASS(μ'') = HOURGLASS(s_2, s_3) by $O(1)$ common-tangent computations, and thus each node μ contains $O(1)$ tangent segments. Again, by segment-tree arguments, the total number of nodes in *ltree*s and *rtree*s is $O(n \log n)$, hence the total number of tangent segments in S is $O(n \log n)$. Also, each segment of S is stored $O(1)$ times in the data structure, since it can have representatives in an allocation node (for an edge fragment), in the highest open hourglass, and in the highest monotone hourglass, and there may be two such nodes for each segment (recall that edges have two "sides," and the corresponding nodes in the paired *ltree* and *rtree* may have duplicate information). We conclude that the secondary structures are a collection of balanced trees with a total of $O(n \log n)$ nodes, and hence use total space $O(n \log n)$. □

Query operations PATHLENGTH(q_1, q_2, r) and PATH(q_1, q_2, r) are performed as follows:

1. Find the trapezoids τ_1 and τ_2 of the trapezoidal decomposition of r containing q_1 and q_2, using the point-location machinery of §5. Let s_1 and s_2 be the splitters on the boundary of τ_1 and τ_2, such that q_1 and q_2 are on opposite sides of SLEEVE(s_1, s_2).
2. Create the solid path P for SLEEVE(s_1, s_2) (P is the path between edges $\delta(s_1)$ and $\delta(s_2)$ of $\delta(r)$), by means of *evert*($\delta(s_1)$) and *expose*($\delta(s_2)$). The secondary structure stored at the root of *lthread*(P) (or *rthread*(P)) yields a representation of HOURGLASS(s_1, s_2).

Given the representation of HOURGLASS(s_1, s_2), after computing in time $O(\log n)$ the tangents from q_1 and q_2 to the appropriate funnels, we can answer PATHLENGTH(q_1, q_2, r) and PATH(q_1, q_2, r) in time $O(1)$ and $O(k)$, respectively (where k is the number of edges of the shortest path reported). Finally, we conceal the path exposed in step 2 to satisfy the weight invariant.

Regarding updates, we have the following lemma (see the example in Fig. 9).

LEMMA 4.2. *An elementary split or join of two thread trees in the normalization structure augmented with the hull structure takes time* $O(\log n)$.

Proof. After an elementary split or join of two solid thread trees, we need to update only the secondary data structures of their roots. Since such data structures represent the hourglasses of the corresponding sleeves, they can be updated in $O(\log n)$ time (see Fig. 9). Note that for a nonmonotone solid path the updates are limited to its leftmost or rightmost monotone subpath. For this reason it is sufficient to store only the length of the string in the nodes of the upper levels of the *lthread* and *rthread* trees. □

As a consequence, splitting a solid path or joining two solid paths takes time $O(\log^2 n)$. Note that parting or pairing *ltree*(Q) and *rtree*(Q) of a monotone path Q (because of an edge insertion or deletion in the corresponding sleeve) takes $O(1)$ time. The lemma below follows from Lemmas 3.8 and 4.2.

LEMMA 4.3. *Queries* PATHLENGTH(q_1, q_2, r) *and* PATH(q_1, q_2, r) *are performed in time* $O(\log^3 n)$ *and* $O(\log^3 n + k)$, *respectively, where k is the number of edges of the shortest path reported.*

Now, we discuss how operation INSERTVERTEX(v, e; e_1, e_2) affects the new data structure. First, we insert a new node $y(v)$ into $\mathcal{Y}$. Let μ be one of the two nodes in the *ltree* and *rtree*

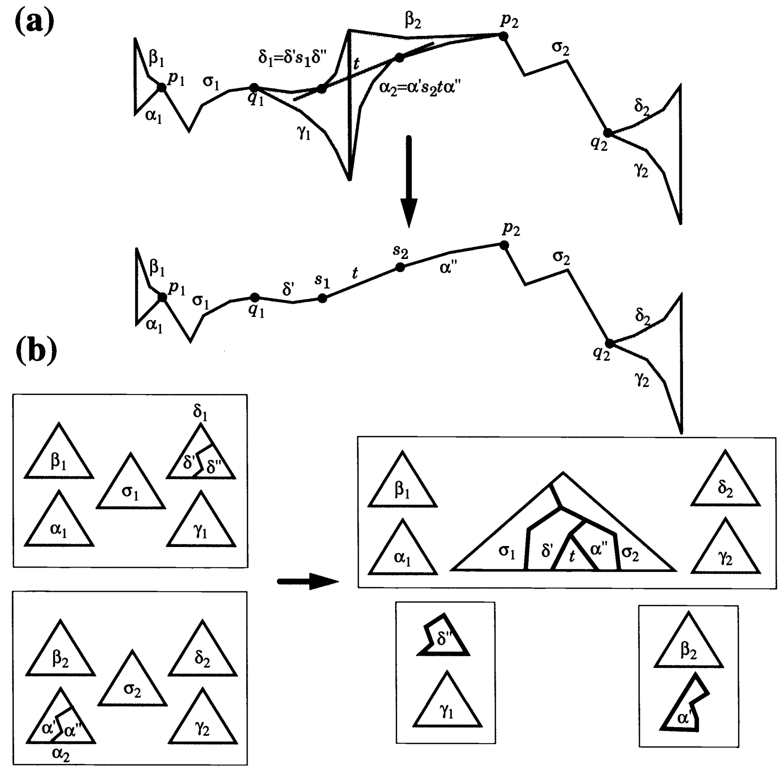


FIG. 9. *Example of update of the secondary structures in an elementary join of two solid paths.* (a) *Geometric construction of the hourglass.* (b) *Construction of the representation of the root hourglass by means of split and join operations on the chain trees in the repr·esentation of the hourglasses of the children nodes.*

that stores the fragment of edge e where v is inserted, and let y be the corresponding node of $\mathcal{Y}$. Before the insertion of v, there is a pointer from μ to y but no back pointer from y to μ, since μ is a leaf of a pruned tree. After the insertion of v, the fragment of e stored in μ is further partitioned into $O(\log n)$ fragments (but the total number of fragments of e is still $O(\log n)$) according to the subtree of $\mathcal{Y}$ rooted at y ($y(v)$ has already been inserted into this subtree); we allocate these edge fragments into a new tree T_μ, expand leaf μ to T_μ, establish a pointer from y to μ, and rename all fragments of e to e_1 or e_2 appropriately.

The insertion of $y(v)$ into $\mathcal{Y}$ may cause rebalancing operations in $\mathcal{Y}$ carried out by means of rotations. A rotation between a node y' and its child y'' implies that horizontal cuts at y'' now take priority over horizontal cuts at y'. It is easy to see that the rotation only affects the subtrees of the *ltree*s and *rtree*s rooted at the nodes pointed to by y'. We rebuild such subtrees from scratch, which can be done in time proportional to their size. Note that prior to the rotation, a leaf μ of *ltree* or *rtree* corresponding to y' stores an edge fragment that spans the canonical vertical interval I of y', and thus μ is not affected by the rotation (except that after the rotation we have to redirect the original pointer of μ to y' so that it now points to y'', since y'' now corresponds to I). Since there may be a large number of such leaves μ that do not require rebuilding, we do not establish a back pointer from y' to leaf μ in our data structure (as we have already seen), so that inefficient checking for the necessity of rebuilding is avoided. Also, the redirection of all pointers of leaves μ from y' to y'' can be done efficiently when we rotate y' with y'': we switch the contents of the *physical* nodes y' and y'' to interchange the roles of the physical nodes y' and y'' (and then carry out the rotation appropriately by $O(1)$ elementary splits and joins), so that all these pointers are effectively redirected, though no actual changes are made to the pointers. Now we show that the rebuilding of the subtrees of *ltree*s and *rtree*s caused by a rotation in $\mathcal{Y}$ can be performed efficiently.

LEMMA 4.4. *Let y be a node of $\mathcal{Y}$ whose subtree has ℓ leaves. The subtrees of ltrees and rtrees with the hull structure appended whose roots are pointed to by node y have total size $O(\ell \log \ell)$ and can be built in time $O(\ell \log \ell)$.*

Proof. The subtree of $\mathcal{Y}$ rooted at y has exactly $2\ell - 1$ nodes. Thus there are $O(\ell)$ vertices inside the canonical vertical interval I of node y. The leaves of the subtree rooted at a node pointed by y store the edge fragments that are inside I but do not span I. Hence, the edges contributing to such fragments must be incident on some vertex inside I. Since each vertex has bounded degree, there are $O(\ell)$ such edges. Also, since the subtree of $\mathcal{Y}$ has height $O(\log \ell)$, each such edge has $O(\log \ell)$ fragments inside I. We conclude that the total number of leaves in the subtrees rooted at node pointed by y is $O(\ell \log \ell)$, and hence their total size is also $O(\ell \log \ell)$. □

By the properties of BB[α]-trees, we derive the following lemma.

LEMMA 4.5. *The amortized rebalancing time of the Y-tree $\mathcal{Y}$ in a sequence of update operations is $O(\log^2 n)$.*

We conclude the following.

THEOREM 4.6. *Shortest-path queries* PATHLENGTH(q_1, q_2, r) *and* PATH(q_1, q_2, r) *in an n-vertex connected planar map can be performed in worst-case time $O(\log^3 n)$ and $O(\log^3 n+k)$, respectively (where k is the number of edges of the shortest path reported), using a fully dynamic data structure that uses space $O(n \log n)$ and supports updates of the map in time $O(\log^3 n)$ (amortized for vertex updates).*

Remark. In a concrete situation where vertices are a priori restricted to a fixed set of ordinates, tree $\mathcal{Y}$ is static; if we then implement the trees *ltree* and *rtree* by means of *contracted binary trees* [27] of depth $< \log |Y|$ (whose maintenance requires no rotation), then the update times become $O(\log^2 n \log |Y|)$, in the worst case.

The following arc two additional types of queries that can be supported by the described data structure without any modification.

TRAILLENGTH$(q_1, q_2 | e_1, \ldots, e_\ell)$. Allowing edges $e_1, \ldots, e_\ell$ to be deleted, are points q_1 and q_2 reachable to each other? If so, then return the length of the shortest path.

TRAIL$(q_1, q_2 | e_1, \ldots, e_\ell)$. Allowing edges $e_1, \ldots, e_\ell$ to be deleted, are points q_1 and q_2 reachable to each other? If so, then return the shortest path.

An immediate application is that viewing the edges of the map as walls, we are allowed to put doors on edges $e_1, \ldots, e_\ell$. Can a pointlike robot at position q_1 reach position q_2? If so, then report the shortest path or its length.

Clearly, by using REMOVEEDGE, point-location query (see §5), PATHLENGTH or PATH, and INSERTEDGE operations, queries TRAILLENGTH and TRAIL can be answered in worst-case time $O((\ell + 1) \log^3 n)$ and $O((\ell + 1) \log^3 n + k)$, respectively, where k is the number of edges of the shortest path reported.

5. Point location. In this section, we consider the problem of answering point-location queries.

LOCATE(q). Find the region containing query point q. If q is on an vertex or edge, then return that vertex or edge.

Our dynamic point-location data structure is inspired by the static trapezoid method [23] and its dynamic version for monotone maps [8]. It uses the normalization and hull structures as the underpinning of update operations. Queries are instead performed in a location structure, a binary tree called *trapezoid tree*.

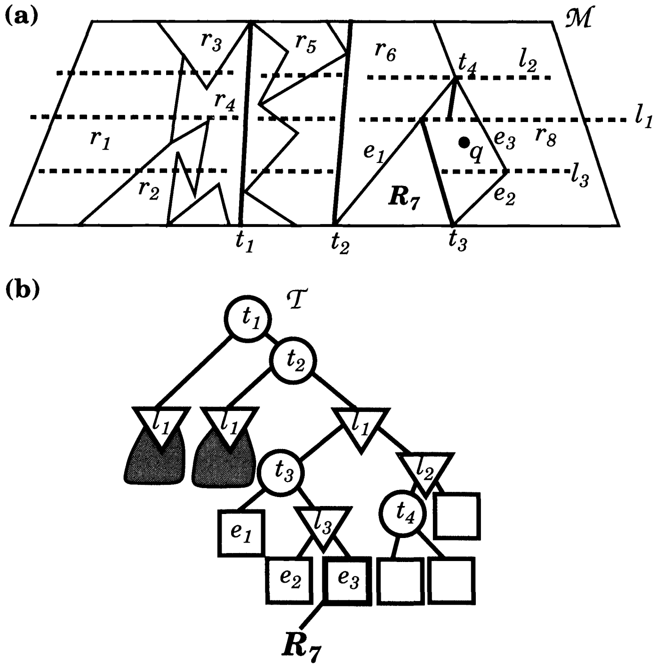


FIG. 10. *Example of the construction of trapezoid tree $\mathcal{T}$ for map $\mathcal{M}$.* (a) *Recursive decomposition of $\mathcal{M}$ by vertical and horizontal cuts.* (b) *Trapezoid tree $\mathcal{T}$ associated with the decomposition in part* (a).

The trapezoid tree defines a binary partition of the plane obtained by means of vertical and horizontal cuts. It differs in many substantial aspects from the trapezoid trees used in [8], [23], the most striking difference being that it is not balanced.

The trapezoid tree $\mathcal{T}$ for map $\mathcal{M}$ is based on the *Y-tree* $\mathcal{Y}$ (see §4) and on the normalization of $\mathcal{M}$ as reflected by the normalization structure (see §3). We view the unnormalized map $\mathcal{M}$ as a trapezoid with its sides at infinity. If a trapezoid τ contains more than a single edge fragment in its interior, we recursively decompose it into trapezoids whose vertical spans are canonical vertical intervals, according to the following rules (see Fig. 10):

Vertical cut. If τ is a coupler or is vertically spanned by a monotone subpath Q and the hourglass H of SLEEVE(Q) is open, we decompose τ by one of the supporting tangents t of H.

Horizontal cut. If no vertical cut is possible, then we decompose τ by cutting it along the horizontal line at the y-coordinate associated with the (unique) allocation node of τ in $\mathcal{Y}$.

Note that a vertical cut always takes priority over a horizontal cut. If more vertical cuts are possible, their order is arbitrary. We represent the above decomposition of $\mathcal{M}$ by means of a binary tree $\mathcal{T}$ (see Fig. 10). Each node of $\mathcal{T}$ is associated with a trapezoid τ of the decomposition and the partitioning object (a tangent or a horizontal line) of τ, and stores the representation of such object. Nodes of $\mathcal{T}$ are classified into three categories (and the association): a ○-*node* (a vertical cut), a ∇-*node* (a horizontal cut), and a □-*node* (a terminal trapezoid of the decomposition and its edge fragment).

The above decomposition process is closely related to the one induced by the segment tree. In particular, the leaves of $\mathcal{T}$ are in one-to-one correspondence with the fragments of the edges of $\mathcal{M}$, so that tree $\mathcal{T}$ has $O(n \log n)$ leaves. Since each node stores a constant amount of information, we have that the space requirement of the trapezoid tree $\mathcal{T}$ is $O(n \log n)$.

It is clear that a point-location query LOCATE(q) can be performed by traversing a root-to-leaf path in $\mathcal{T}$, where at each internal node μ we branch left or right depending on the discrimination of the query point q with respect to the partitioning object stored at μ. Indeed, the leaf reached identifies an edge that is first hit by a horizontal ray through q. Since we did not impose any balance requirement on $\mathcal{T}$, the query time could be linear in the worst case.

To speed up queries, we implement $\mathcal{T}$ as a dynamic tree [30], i.e., $\mathcal{T}$ is decomposed into solid paths (which should not be confused with the solid paths in the normalization structure), connected by dashed arcs (see Fig. 11). Each solid path is associated with a path tree, implemented as a biased search tree [3]. Note that the sequence of nodes of a solid path of $\mathcal{T}$ identifies a sequence of nested trapezoids. For example, in path tree $T(P_1)$ of Fig. 11(c), leaf t_1 identifies the trapezoid of the entire map $\mathcal{M}$, and leaf t_4 identifies the trapezoid whose right side is at infinity and whose other sides are t_2, l_1, and l_2. A point-location query starts at the root of the path tree of the topmost solid path of $\mathcal{T}$ (e.g., the root of $T(P_1)$ in Fig. 11(c)). At a given internal node η of a path tree, we consider the rightmost node ζ in the left subtree of η (readily available given thread pointers). We discriminate q against the trapezoid τ of ζ and go to the left or right child of η according to whether q is inside or outside τ (recall that a solid path is stored bottom-to-top in the left-to-right leaves of its path tree). When we reach a leaf of a path tree (which represents a node μ of $\mathcal{T}$), we always exit on a dashed arc, and we always know the exit except for the case of the last node of the solid path, in which case we go to its left or right child by discriminating q against the partitioning object of μ. For example, in Fig. 11(c), when we reach leaf l_1 of $T(P_1)$, we know that the next node to visit is the root of $T(P_2)$, since that is the only exit; when we reach leaf l_3 of $T(P_2)$, we discriminate q with l_3 and move down right to $T(P_5)$ by the fact that q is above l_3. By this process, we will finally reach a leaf of a path tree with no exit (representing a leaf of $\mathcal{T}$), which identifies an edge of the region containing q.

Using biased search trees [3] as the standard implementation of path trees, we have the following lemma.

LEMMA 5.1. *The time complexity for a point-location query is* $O(\log n)$.

Proof. Let $(\nu_1, \mu_1), \ldots, (\nu_\ell, \mu_\ell)$ be the sequence of dashed arcs traversed by the query algorithm, with ν_i the parent of μ_i. (Note that μ_ℓ is the leaf reached by the query algorithm.) Also, let μ_0 be the root of $\mathcal{T}$. Since the path trees are implemented as biased search trees, we have that the number of nodes visited in the solid path of ν_i is at most $\log(\mathit{weight}(\mu_{i-1})/\mathit{weight}(\nu_i)) + 2$. Hence, the time complexity of a point-location query is $O(\sum_{i=1}^{\ell} \log(\mathit{weight}(\mu_{i-1})/\mathit{weight}(\nu_i)))$. Since $\mathit{weight}(\mu_i) < \mathit{weight}(\nu_i)$, the above sum telescopes, and we have that a point-location query takes time $O(\log n)$. □

To perform update operations, we establish bidirectional links between the trapezoid tree and the normalization structure. Let μ be a node of $\mathcal{T}$. We have the following:

- If μ is a $\bigcirc$-node, let Q' be the subpath of a monotone path Q associated with the vertical cut at μ (i.e., the sleeve of Q' spans the trapezoid of μ and has an open hourglass). We establish pointers between μ and the nodes of $\mathit{ltree}(Q)$ and $\mathit{rtree}(Q)$ associated with Q'.
- If μ is a ∇-node, let Q' and R' be subpaths of monotone paths Q and R such that Q' and R' span the leftmost and rightmost regions in the trapezoid of μ. We establish pointers between μ and the nodes of $\mathit{ltree}(R)$ and $\mathit{rtree}(Q)$ associated with R' and Q', respectively. Also, we establish a back pointer from the allocation node y of μ in $\mathcal{Y}$ to μ.
- If μ is a $\Box$-node, we establish pointers between μ and the two nodes in the normalization structure associated with the same edge fragment.

Note that every node of a thread tree associated with an open hourglass is pointed to by exactly two nodes of $\mathcal{T}$.

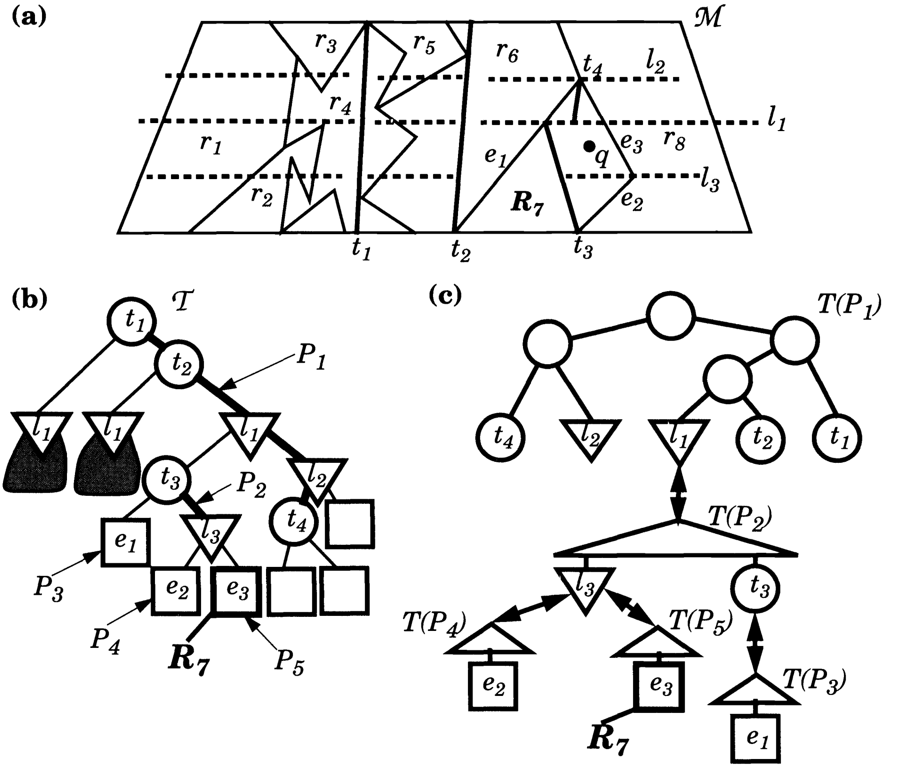


FIG. 11. *Representing trapezoid tree $\mathcal{T}$ by a dynamic tree.* (a) *The same decomposition of $\mathcal{M}$ as in Fig.* 10(a). (b) *Decomposing trapezoid tree $\mathcal{T}$ of Fig.* 10(b) *into solid paths P_1, P_2, ...* (c) *Actual data structure representing $\mathcal{T}$, where $T(P_i)$ is the path tree for solid path P_i in $\mathcal{T}$. The left-to-right leaves of $T(P_i)$ represent bottom-to-top nodes of P_i, which in turn correspond to smaller-to-bigger nested trapezoids.*

Now, we discuss how update operations affect the trapezoid tree. Since the decomposition described by $\mathcal{T}$ is determined by the monotone paths, we update the trapezoid tree whenever monotone paths are changed in the normalization structure. We only need to consider the effects on the trapezoid tree of elementary splits, joins, partings, and pairings of monotone paths. Each such elementary operation in the normalization structure corresponds to performing $O(1)$ link and cut operations in the trapezoid tree. Details are shown in Figs. 12 and 13. Link and cut operations are performed in $O(\log n)$ time by standard dynamic tree algorithms. Regarding vertex insertions, a rotation at a node y in the *Y-tree* $\mathcal{Y}$ caused by a vertex update is handled by rebuilding the subtrees of $\mathcal{T}$ whose roots are ∇-nodes pointed by y. With an argument analogous to the one of Lemma 4.4, we can prove the following lemma.

LEMMA 5.2. *Let y be a node of $\mathcal{Y}$ whose subtree has ℓ leaves. Then the subtrees of $\mathcal{T}$ whose roots are pointed to by y have total size $O(\ell \log \ell)$ and can be built in time $O(\ell \log \ell)$.*

Hence the amortized cost of rebalancing $\mathcal{Y}$ in a sequence of updates is $O(\log^2 n)$. We conclude the following.

THEOREM 5.3. *Point-location queries* LOCATE(q) *in an n-vertex connected planar map can be performed in worst-case time $O(\log n)$ using a fully dynamic data structure that uses space $O(n \log n)$ and supports updates of the map in time $O(\log^3 n)$ (amortized for vertex updates).*

Note that query LOCATE(q) is used in the update of the hull structure.

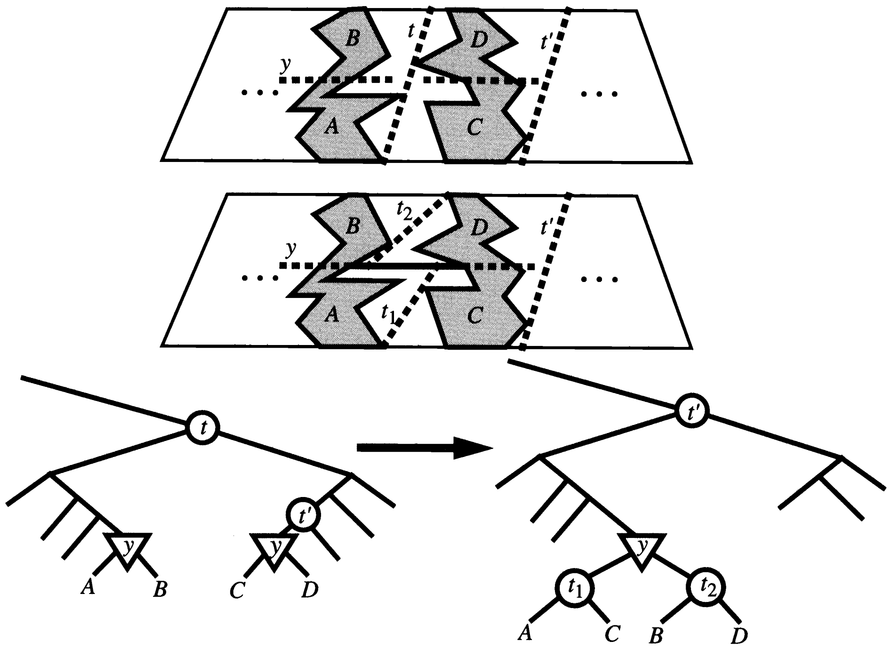


FIG. 12. *Update of the trapezoid tree in consequence of an elementary split of a monotone path in the normalization structure.*

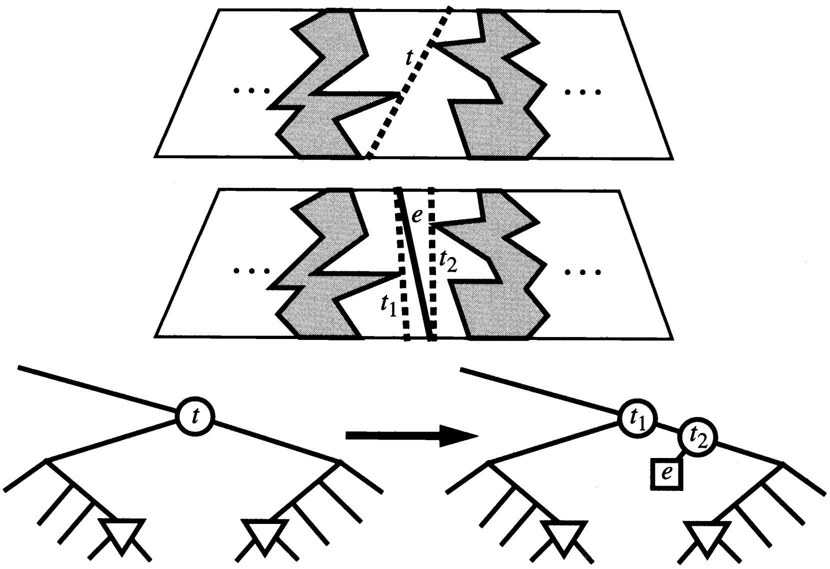


FIG. 13. *Update of the trapezoid tree caused by parting a monotone path in the normalization structure because of an edge insertion.*

6. Ray shooting. In this section, we consider the problem of performing ray-shooting queries of the following type:

SHOOT(q, d). Find the first vertex or edge hit by a query ray (q, d) in direction d originating at point q.

We show that the dynamic point-location data structure in the previous section also supports ray-shooting queries in worst-case time $O(\log^3 n)$. Without loss of generality, assume that (q, d) is oriented upwards. The ray-shooting algorithm is as follows.

First, we perform LOCATE(q) to determine the region r containing q. If q lies on a vertex or edge, an infinitesimal perturbation of q in direction d enables us to find the first region r entered by the ray. Query LOCATE(q) also identifies the monotone sleeve SLEEVE(Q) of r containing q and the splitter s_1 of SLEEVE(Q) immediately below q. We find the first intersection q' of (q, d) with the boundary of SLEEVE(Q). If q' is on a vertex or edge of r, then we report q' and stop; else (q' is on a lid of SLEEVE(Q)) we apply the algorithm recursively to the new ray (q', d).

We find the first intersection q' of (q, d) with the boundary of SLEEVE(Q) by the process below:

1. Find the topmost splitter s_2 in SLEEVE(Q) such that HOURGLASS(s_1, s_2) is open, by means of $O(\log n)$ elementary splits and joins of subpaths of Q that yield a new monotone path R such that SLEEVE(s_1, s_2) = SLEEVE(R). Note that the boundary of SLEEVE(R) is part of the boundary of SLEEVE(Q) except for possibly s_1 and s_2, where s_2 is part of the boundary of SLEEVE(Q) if and only if s_2 is the top lid of SLEEVE(Q).
2. Find the first intersection p of (q, d) with the boundary of SLEEVE(R).
3. If p is not on s_2, or if p is on s_2 but s_2 is the top lid of SLEEVE(Q), then p is on the boundary of SLEEVE(Q) and thus the desired intersection q'. Return p and stop.
4. Else (p is on s_2 and s_2 is not the top lid of SLEEVE(Q)), set $s_1 := s_2$, $q := p$, and go to step 1. Note that this situation can occur at most twice, since s_2 is the topmost splitter above s_1 such that HOURGLASS(s_1, s_2) is open, and any straight line can completely go through at most one such hourglass, with the bottom and top portions of the line possibly in the two (below and above) adjacent hourglasses (see Fig. 14).

In step 2, the first intersection p of (q, d) with the boundary of SLEEVE(R) can be found by a binary search in the trees *ltree*(R) and *rtree*(R) as follows: at a current node μ with children μ' and μ'', where CHAIN(μ') is below CHAIN(μ''), we determine the intersection of (q, d) with the convex hull of CHAIN(μ). If the intersection is on a real edge or on a lid, then we are done. Else it is on a (fictitious) convex hull edge; we then compute the complete convex hulls of CHAIN(μ') and CHAIN(μ''), and repeat the process on μ' or on μ'' depending on whether or not (q, d) intersects with the convex hull of CHAIN(μ'), respectively.

The computation of point q' can be done in $O(\log^2 n)$ time: step 1 performs $O(\log n)$ elementary joins and splits of solid subpaths of Q, each in $O(\log n)$ time by Lemma 4.2; step 2 takes $O(\log^2 n)$ time, with $O(\log n)$ time on each node visited during the binary search; finally, the steps are executed at most three times by step 4. The number of recursive calls to compute a sequence of such points q' is $O(\log n)$, since the query ray intersects $O(\log n)$ lids by Corollary 3.3. At the end, we conceal the path of $\Delta(r)$ traversed by the query ray to restore the weight invariant. We conclude with the following theorem.

THEOREM 6.1. *Ray-shooting queries* SHOOT(q, d) *in an n-vertex connected planar map can be performed in worst-case time $O(\log^3 n)$ using a fully dynamic data structure that uses space $O(n \log n)$ and supports updates of the map in time $O(\log^3 n)$ (amortized for vertex updates).*

Theorem 6.1 also provides the capability of checking the validity of an edge insertion, i.e., whether the new edge does not intersect the current edges of the map. Moreover, as a corollary, we can perform stabbing queries, namely, determine the k edges of the map intersected by a query segment, in time $O((k + 1) \log^3 n)$.

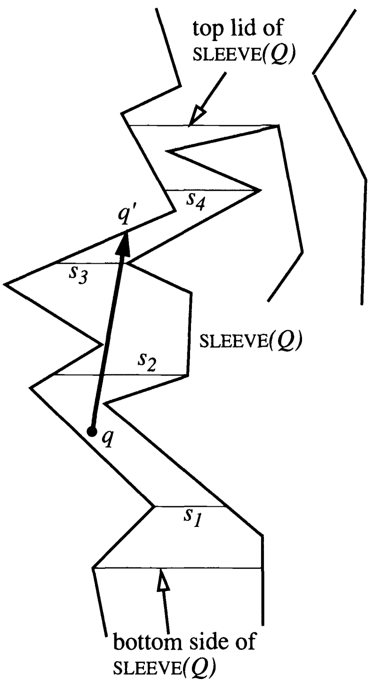


FIG. 14. *The situation in step* 4 *of the process for computing* q' *can occur at most twice. For* $i = 1, 2, 3$, s_{i+1} *is the topmost splitter above* s_i *such that* HOURGLASS(s_i, s_{i+1}) *is open. As shown, the situation of step* 4 *occurrs twice when* (q, d) *hits* s_2 *and* s_3, *respectively. Note that* (q, d) *cannot reach* s_4, *or otherwise* HOURGLASS(s_2, s_4) *would be open and* s_3 *would not be the topmost splitter above* s_2 *such that* HOURGLASS(s_2, s_3) *is open.*

REFERENCES

[1] P. K. AGARWAL AND M. SHARIR, *Applications of a new partition scheme*, Discrete Comput. Geom., 9 (1993), pp. 11–38.

[2] H. BAUMGARTEN, H. JUNG, AND K. MEHLHORN, *Dynamic point location in general subdivisions*, in Proc. 3rd ACM–SIAM Symposium on Discrete Algorithms, 1992, pp. 250–258.

[3] S. W. BENT, D. D. SLEATOR, AND R. E. TARJAN, *Biased search trees*, SIAM J. Comput., 14 (1985), pp. 545–568.

[4] G. BILARDI AND F. P. PREPARATA, *Probabilistic analysis of a new geometric searching technique*, unpublished manuscript, 1981.

[5] B. CHAZELLE AND L. J. GUIBAS, *Visibility and intersection problems in plane geometry*, Discrete Comput. Geom., 4 (1989), pp. 551–581.

[6] S. W. CHENG AND R. JANARDAN, *New results on dynamic planar point location*, SIAM J. Comput., 21 (1992), pp. 972–999.

[7] ———, *Space efficient ray shooting and intersection searching: Algorithms, Dynamizations, and Applications*, in Proc. 2nd ACM–SIAM Symposium on Discrete Algorithms, 1991, pp. 7–16.

[8] Y.-J. CHIANG AND R. TAMASSIA, *Dynamization of the trapezoid method for planar point location in monotone subdivisions*, Internat. J. Comput. Geom. Appl., 2 (1992), pp. 311–333.

[9] ———, *Dynamic algorithms in computational geometry*, Proc. Institute for Electrical and Electronics Engineering, G. Toussaint, ed., 80 (1992), pp. 1412–1434.

[10] D. P. DOBKIN AND R. LIPTON, *Multidimensional searching problems*, SIAM J. Comput., 5 (1976), pp. 181–186.

[11] M. I. EDAHIRO, I. KOKUBO AND T. ASANO, *A new point-location algorithm and its practical efficiency—comparison with existing algorithms*, ACM Trans. Graphics, 3 (1984), pp. 86–109.

[12] H. EDELSBRUNNER, L. J. GUIBAS, AND J. STOLFI, *Optimal point location in a monotone subdivision*, SIAM J. Comput., 15 (1986), pp. 317–340.

[13] O. FRIES, *Zerlegung einer planaren unterteilung der ebene und ihre anwendungen*, M.S. thesis, Institut für Angnewandte Mathematic und Informatik, Universitüt Saarlandes, Saarbrücken, Germany, 1985.

[14] O. FRIES, K. MEHLHORN, AND S. NÄHER, *Dynamization of geometric data structures*, in Proc. 1st ACM Symposium on Computational Geometry, 1985, pp. 168–176.

[15] M. T. GOODRICH AND R. TAMASSIA, *Dynamic trees and dynamic point location*, in Proc. 23rd ACM Symposium on Theory of Computing, 1991, pp. 523–533.

[16] L. J. GUIBAS AND J. HERSHBERGER, *Optimal shortest path queries in a simple polygon*, J. Comput. System Sci., 39 (1989), pp. 126–152.

[17] D. G. KIRKPATRICK, *Optimal search in planar subdivisions*, SIAM J. Comput., 12 (1983), pp. 28–35.

[18] D. T. LEE AND F. P. PREPARATA, *Location of a point in a planar subdivision and its applications*, SIAM J. Comput., 6 (1977), pp. 594–606.

[19] ———, *Euclidean shortest paths in the presence of rectilinear barriers*, Networks, 14 (1984), pp. 393–410.

[20] K. MEHLHORN, *Data Structure and Algorithms* 1: *Sorting and Searching*, Springer-Verlag, Heidelberg, Germany, 1984, pp. 189–199.

[21] M. H. OVERMARS, *Range searching in a set of line segments*, in Proc. 1st ACM Symposium on Computational Geometry, 1985, pp. 177–185.

[22] M. H. OVERMARS AND J. VAN LEEUWEN, *Maintenance of configurations in the plane*, J. Comput. System Sci., 23 (1981), pp. 166–204.

[23] F. P. PREPARATA, *A new approach to planar point location*, SIAM J. Comput., 10 (1981), pp. 473–483.

[24] F. P. PREPARATA AND M. I. SHAMOS, *Computational Geometry*: *An Introduction*, Springer-Verlag, New York, 1985.

[25] F. P. PREPARATA AND R. TAMASSIA, *Fully dynamic point location in a monotone subdivision*, SIAM J. Comput., 18 (1989), pp. 811–830.

[26] ———, *Dynamic planar point location with optimal query time*, Theoret. Comput. Sci., 74 (1990), pp. 95–114.

[27] F. P. PREPARATA, J. VITTER, AND M. YVINEC, *Computation of the axial view of a set of isothetic parallelepipeds*, ACM Trans. Graphics, 9 (1990), pp. 278–300.

[28] J. H. REIF AND S. SEN, *An efficient output-sensitive hidden-surface removal algorithm and its parallelization*, in Proc. 4th ACM Symposium on Computational Geometry, 1988, pp. 193–200.

[29] N. SARNAK AND R. E. TARJAN, *Planar point location using persistent search trees*, Commun. Assoc. Comput. Mach., 29 (1986), pp. 669–679.

[30] D. D. SLEATOR AND R. E. TARJAN, *A data structure for dynamic trees*, J. Comput. System Sci., 24 (1983), pp. 362–381.

[31] R. TAMASSIA, *An incremental reconstruction methodfor dynamic planar point location*, Inform. Process. Lett., 37 (1991), pp. 79–83.

SIAM J. COMPUT.
Vol. 25, No. 2, pp. 235–251, April 1996

APPROXIMATE MAX-FLOW MIN-(MULTI)CUT THEOREMS AND THEIR APPLICATIONS*

NAVEEN GARG[†], VIJAY V. VAZIRANI[†], AND MIHALIS YANNAKAKIS[‡]

Abstract. Consider the multicommodity flow problem in which the object is to maximize the sum of commodities routed. We prove the following approximate max-flow min-multicut theorem:

$$\frac{\text{min multicut}}{O(\log k)} \leq \text{ max flow } \leq \text{ min multicut},$$

where k is the number of commodities. Our proof is constructive; it enables us to find a multicut within $O(\log k)$ of the max flow (and hence also the optimal multicut). In addition, the proof technique provides a unified framework in which one can also analyse the case of flows with specified demands of Leighton and Rao and Klein et al. and thereby obtain an improved bound for the latter problem.

Key words. approximation algorithm, multicommodity flow, minimum multicut

AMS subject classifications. 68Q25, 90B10

1. Introduction.

Much of flow theory, and the theory of cuts in graphs, is built around a single theorem—the celebrated max-flow min-cut theorem of Fort and Fulkerson [FF] and Elias, Feinstein, and Shannon [EFS]. The power of this theorem lies in that it relates two fundamental graph-theoretic entities via the potent mechanism of a min-max relation.

The importance of this theorem has led researchers to seek its generalization to the case of *multicommodity flow*. In this setting, each commodity has its own source and sink, and the object is to maximize the sum of the flows subject to capacity and flow conservation requirements. The notion of a *multicut* generalizes that of a cut and is defined as a set of edges whose removal disconnects each source from its corresponding sink. Clearly, maximum multicommodity flow is bounded by minimum multicut; the question is whether equality holds. This can be established for some special cases, e.g., if there are only two commodities [Hu]; however, one can construct very simple examples to show that equality does not hold in general. Consider a tree of height one with three leaves. Each pair of leaf vertices form the source–sink pair for a commodity. All edges have unit capacities. The max flow in this graph is $\frac{3}{2}$, whereas the minimum multicut is 2.

Why does the theorem hold for a single commodity, and why does the generalization fail? For an explanation, consider the LP formulation of the maximum multicommodity flow problem. As shown in §5, the dual of this is the LP relaxation of the minimum multicut problem, i.e., the optimal *integral* solution to the dual is the minimum multicut. In general, the vertices of the dual polyhedron are not integral. However, for the case of a single commodity, they are integral (see [GV] for an exact characterization), and the max-flow min-cut theorem is simply a consequence of the LP-duality theorem. For the multicommodity case, the LP-duality theorem shows only that maximum flow is equal to the minimum fractional (i.e., relaxed) multicut.

In this situation, the best one can hope for is an approximate max-flow min-cut theorem. In ground-breaking work, Leighton and Rao [LR] gave the first such

* Received by the editors January 19, 1993; accepted for publication (in revised form) August 5, 1994.

[†] Department of Computer Science and Engineering, Indian Institute of Technology, Delhi, India.

[‡] AT&T Bell Laboratories, Murray Hill, NJ 07974 (mihalis@research.att.com).

theorem. Let us consider a second formulation of the multicommodity flow problem that has also been widely studied in the past. In this formulation, a demand, D_i, is specified for each commodity, i. The object is to determine the maximum number, f, called *throughput*, such that fD_i amount of each commodity i can be routed simultaneously, subject to capacity and conservation constraints. (Equivalently, the object is to determine the minimum number, u, such that if the capacity of each edge is multiplied by u, then all the demands can be simultaneously satisfied. Clearly, at optimality $f = \frac{1}{u}$.) The analogue of a minimum cut in this case is a sparsest cut into two parts, one that minimizes the ratio of capacity of the cut to the demand across the cut. Let α be this minimum. Clearly, $f \leq \alpha$, and once again equality does not generally hold.

Leighton and Rao considered a special case of the above-stated formulation, called uniform multicommodity flow, in which there is a commodity corresponding to each pair of vertices and all the demands are unity. They proved the following approximate max-flow min-cut theorem:

$$\frac{\alpha}{O(\log n)} \leq f \leq \alpha,$$

where n is the number of vertices in the graph. Subsequently, Klein et al. [KARR] managed to attack the arbitrary demands problem, and proved that

$$\frac{\alpha}{O(\log C \log D)} \leq f \leq \alpha,$$

where C is the sum of capacities of all edges and D is the sum of all demands. However, one restriction they impose is that all capacities and demands be integral. The lower bound was later improved to $\alpha/O(\log n \log D)$ by Tragoudas [Trag]. [LR], [KARR], and [Trag] also give polynomial-time algorithms for finding an approximation to the sparsest cut, the factors being $O(\log n)$ and $O(\log C \log D)$ (or $O(\log n \log D)$), respectively.

We address the first version of the multicommodity flow problem, henceforth referred to as the *maximum multicommodity flow* problem, and prove the following approximate max-flow min-multicut theorem:

$$\frac{M}{O(\log k)} \leq f \leq M,$$

where f is the max flow, M is the minimum multicut, and k is the number of commodities. We also show that our theorem is tight up to a constant factor, and we give a polynomial-time algorithm for finding a multicut within $O(\log k)$ of the optimal fractional, and therefore also of the integral multicut.

Our general approach is similar to that of Leighton and Rao [LR] and Klein et al. [KARR]. We consider the LP-relaxation of the minimum multicut problem and use its optimal solution to define a graph with distance labels on the edges. Starting from a source or a sink, we grow a region in this graph until we find a cut of small enough capacity separating the root from its mate. The region is removed and the process is repeated. Our method differs in several respects from previous methods. It dispenses with the discretization of the edge distances and employs a technique that leads to quicker termination of the region growth process and thus produces a better bound on the capacity of the cut. These techniques are encapsulated in the two region-growing lemmas of §4, which use the idea of packing cuts to grow regions.

Our analysis is also useful in the demands version of the multicommodity flow problem. It establishes a unified framework in which simpler proofs of the theorems of Leighton and Rao and Klein et al. can be given by dispensing with tokenizing distances. We also avoid the dynamic resetting of the parameters for region growing and restarting the procedure. In both cases we use heavily ideas from the original papers. Using our lemmas, we improve the [KARR] and [Trag] results to

$$\frac{\alpha}{O(\log k \log D)} \leq f \leq \alpha.$$

We also dispense with the restriction that capacities be integral. Furthermore, Plotkin and Tardos [PT] give a method of scaling demands so that the log D factor in these results can be replaced by $\log k$, thus yielding an improved bound of $O(\log^2 k)$ on the gap between f and α.

The following problem is a generalization of the uniform multicommodity flow problem considered by Leighton and Rao and by Tragoudas. It was called the *product multicommodity flow problem* in [LR] and the *complete concurrent flow problem* in [Trag]. In this problem, each vertex has a nonnegative weight $w(v)$ (assumed to be positive integer in [LR] and [Trag] but this is not essential), and there is a commodity for each unordered pair u, v of vertices, with a demand of $w(u)w(v)$. The object again is to maximize the throughput subject to capacity and conservation constraints. Let

$$\alpha = \min_{S \subseteq V} \frac{C_{\nabla(S)}}{w(S)w(\bar{S})},$$

where $w(S)$ in the sum of weights of vertices in S.

Let W be the sum of weights of all vertices, and C be the sum of all edge capacities. Then, Leighton and Rao prove $\alpha/O(\log\min(\frac{W}{w_{\min}}, \frac{C}{c_{\min}})) \leq f \leq \alpha$, where $w_{\min}$ is the weight of the lightest vertex, and $c_{\min}$ is the minimum over all vertices of the sum of capacities of edges incident at the vertex. Tragoudas improves the lower bound to $\alpha/O(\log n)$ [Trag]. Using our techniques, we improve this result to

$$\frac{\alpha}{O(\log k)} \leq f \leq \alpha,$$

where k is the number of vertices having nonzero weight.

The multicut problem finds numerous applications, e.g., in circuit partitioning problems. It was first stated by Hu in 1963 [Hu]. For $k = 1$, the problem coincides with the ordinary min cut problem. For $k = 2$, it can also be solved in polynomial time by two applications of a max-flow algorithm [YKCP]. The problem was proven NP-hard and MAX SNP-hard for any $k \geq 3$ by Dalhaus et al. [DJPSY]. Because of the MAX SNP-hardness, there is no polynomial-time approximation scheme for multicut for $k \geq 3$ (assuming $P \neq NP$) [ALMSS]. Note that, in the demands case, the sparsest cut problem can be solved in polynomial time for fixed k (or $k = O(\log n)$) because in this case we are concerned only with cuts into two parts, and thus we can try all possible partitions of the sources and sinks into two parts and compute the minimum cut for each partition.

Dahlhaus et al. [DJPSY] studied the *multiway cut* (or *multiterminal cut*) problem: given a set of "terminal" vertices T, find a minimum weight set of edges that disconnects every terminal from every other terminal. This is the special case of the multicut problem where there is one commodity for every pair of vertices from the subset T. [DJPSY] gave a factor-of-2 approximation algorithm for this case and showed that it can be used to approximate the general multicut problem within a factor of 2 with a running time that has k in the exponent. Thus the running time is polynomial only for fixed k.

Klein et al. [KARR] used their approximation algorithm for the sparsest cut to give an $O(\log^3 n)$ approximation algorithm for multicut. They also gave some applications of the multicut problem obtaining approximation algorithms with ratio $O(\log^3 n)$ for the following problems: deleting the minimum number of clauses to make a 2CNF $\equiv$ formula satisfiable, deleting the minimum number of edges from a graph to make it bipartite, and a via minimization problem in VLSI. Our improvement for multicut gives us an $O(\log n)$ approximation algorithm for these problems.

2. LP formulations for max multicommodity flow. Given an undirected graph $G = (V, E)$, a capacity function $c : E \rightarrow \Re^+$, and k pairs of vertices (not necessarily distinct) $\{s_i, t_i\}$ $1 \leq i \leq k$, we associate a commodity, i, with the pair $\{s_i, t_i\}$ and designate s_i as the source and t_i as the sink for commodity i. A *multicommodity flow* is a way of simultaneously routing commodities from their sources to sinks, subject to capacity and conservation constraints.

The assumption that each commodity has a single source and a single sink can be made without loss of generality. The more general case where a commodity i may have a set S_i of sources and a set T_i of sinks can be easily reduced to this one by adding a new source s_i with edges to the vertices in S_i and a new sink t_i with edges to T_i.

A multicommodity flow in which the sum of the flows over all the commodities is maximized will be called a *max* (*multicommodity*) *flow*. A *multicut* is defined as a set of edges whose removal disconnects each $\{s_i, t_i\}$ pair. The weight of the multicut is the sum of the capacities of the edges in it. The MULTICUT problem is to find a multicut of minimum weight.

We say that two vertices *share index* i if they form the source–sink pair for commodity i.

Assume that there exist edges (t_i, s_i) in $G, 1 \leq i \leq k$. These edges are special; the only flow allowable on edge (t_i, s_i) is commodity i flowing from t_i to s_i. There are no capacity restrictions on these edges. This allows us to view max flow as a circulation in which the sum of the flows in the edges $(t_i, s_i), 1 \leq i \leq k$, is to be maximized. Let f^l_{ij} denote the flow of commodity l in edge (i, j). The LP formulation of the problem is as follows:

$$\begin{array}{llll}
\text{maximize} & \displaystyle\sum_{i=1}^{k} f^i_{t_i s_i} & & \\
\text{subject to} & \displaystyle\sum_{(j,i)\in E} f^l_{ji} - \sum_{(i,j)\in E} f^l_{ij} \leq 0 & \forall i \in V & \forall l \in [l, \ldots, k], \\
& \displaystyle\sum_{l=1}^{k} f^l_{ij} + \sum_{l=1}^{k} f^l_{ji} \leq c_{ij} & \forall (i,j) \in E - \bigcup_{i=1}^{k}\{(t_i, s_i)\}, & \\
& f^l_{ij} \geq 0 & \forall (i,j) \in E & \forall l \in [1, \ldots, k].
\end{array}$$

The first set of inequalities says that the total flow of each commodity into vertex i is at most the total flow out of it. Note that, if these inequalities hold for each vertex $i \in V$, then in fact they must all hold with equality, thereby implying flow conservation at each node. This is because a deficit in the flow balance at one node must imply a surplus at some other. The second set of inequalities are capacity constraints on the edges; the total flow over all commodities summed in both directions is at most the capacity of the edge.

The dual of this LP is

$$\begin{array}{llll}\text{minimize} & \displaystyle\sum_{(i,j)\in E} d_{ij}c_{ij} & & \\ \text{subject to} & d_{ij} \geq p_i^l - p_j^l & \forall (i,j) \in E - \displaystyle\bigcup_{i=1}^{k}\{(t_i,s_i)\} & \forall l \in [1,\dots,k], \\ & p_{s_i}^i - p_{t_i}^i \geq 1 & \forall l \in [1..k], & \\ & p_i^l \geq 0 & \forall i \in V & \forall l \in [1,\dots,k], \\ & d_{ij} \geq 0 & \forall (i,j) \in E - \displaystyle\bigcup_{i=1}^{k}\{(t_i,s_i)\}. & \end{array}$$

The variable d_{ij} can be viewed as a distance label on the edge (i,j) and p_i^l as the potential corresponding to commodity l on vertex i. Thus the dual problem is an assignment of potentials to vertices and distance labels to edges so that the potential difference (for each commodity) across each edge is no more than the distance label of that edge. Furthermore, the potential difference between the source and the sink for each commodity should be at least 1. These two conditions imply that the distance between each s_i, t_i under the distance label assignment d_{ij} is at least 1. The following LP (with, however, exponentially many constraints) expresses this much more simply.

$$\begin{array}{lll}\text{minimize} & \displaystyle\sum_{e\in E} d_e c_e & \\ \text{subject to} & \displaystyle\sum_{e\in E} d_e q_i^j(e) \geq 1 & \forall q_i^j \\ & d_e \geq 0 & \forall e \in E,\end{array}$$

where q_i^j denotes the jth path in G (under some arbitrary numbering) from s_i to t_i and $q_i^j(e)$ is the characteristic function of this path, i.e., $q_i^j(e) = 1$ if $e \in q_i^j, 0$ otherwise.

Clearly the distance labels of a feasible solution to the first LP give a feasible solution to the second LP with the same objective function. Conversely, given a feasible solution to the second LP compute potentials on the vertices for each commodity as follows:

p_i^l = length of the shortest path from vertex i to the sink for commodity l, under distance labels d_e.

It can be shown that these potentials, together with the distance labels d_e, are a feasible solution to the first LP with the same objective function. Hence the two formulations of the dual are equivalent.

The dual program can now be viewed as an assignment of nonnegative distance labels d_e to edges $e \in E$, so as to minimize $\sum_{e\in E} d_e c_e$, subject to the constraint that each $\{s_i, t_i\}$ pair be at least a unit distance apart. An integral solution to the dual problem corresponds to a multicut; the edges with $d_e = 1$ form a multicut. Hence, the dual is the LP relaxation of the MULTICUT problem.

3. Overview of the algorithm. In this section we will give a high-level description of our algorithm, justifying the steps taken on intuitive grounds.

Our goal is to pick a set of edges of small capacity whose removal separates all s_i, t_i pairs; the total capacity of edges picked should be within a small factor of the max flow (our factor is $O(\log k)$). Clearly, such edges will be bottlenecks for the max flow, so one possibility is to find a max flow using an LP subroutine and start with the set of saturated edges. A better possibility is to find an optimal solution to the dual LP and consider the set of edges having positive distance labels. Notice that, by complementary slackness, $d_e > 0 \Rightarrow \sum_{l=1}^{k} f_{ij}^l + \sum_{l=1}^{k} f_{ji}^l = c_e$, where $e = (i, j)$, i.e., e must be saturated in every max flow. Moreover, the edges $D = \{e | d_e > 0\}$ constitute a multicut.

The entire set D may have a very large capacity; we wish to pick a small capacity subset that is still a multicut. The optimal dual solution is the most cost effective way of picking a fractional multicut. This provides the clue that, for our purpose, edges with large distance labels should be more important than edges with small distance labels. Our algorithm indirectly gives preference to edges having large distance labels. We start by defining the length of edge e in G as d_e. We then find disjoint sets, called regions, such that for each set S, $C_{\nabla(S)} \leq \epsilon. wt(S)$, where $C_{\nabla(S)}$ is the capacity of the cut $(S, \overline{S})$, ϵ is an appropriately chosen parameter, and $wt(S)$ is roughly $\sum c_e d_e$, where the sum is over all edges having at least one endpoint in S. No region contains both source and sink of any pair, and for each commodity either the source or the sink is in some region. Under these conditions, the union of the cuts of the regions is a multicut and has capacity bounded by $2\epsilon F$, where F is the value of the maximum flow.

The overall approach of finding the optimal fractional solution to the dual LP and then growing regions was introduced by Leighton and Rao [LR] for the uniform multicommodity flow problem. The procedure for growing regions is similar to a graph clustering technique first proposed by Awerbuch in [Aw] (for graphs without capacities or lengths on the edges), and is also similar to that used by Leighton and Rao [LR] and Klein et al. [KARR] (for graphs with capacities and lengths) in the context of multicommodity flows. Each region is formed by growing out radially, with respect to the edge lengths d_e, from one of the sources, as in the usual shortest path computation; the region is grown as long as it accumulates weight fast enough. The reason for adopting radial growth is that this maximizes the weight of the region for a given bound on the pairwise distance between vertices in the region. The region-growing process is formally described in the next section. The main differences from [LR] and [KARR] are in the initialization of the process (assignment of initial weights wt for the roots of the regions), the elimination of the discretization of the lengths of the edges, and the use of auxiliary variables associated with the layers of the radial growth. The idea of packing cuts is used for growing the regions and for accounting. This yields simpler proofs, as well as a more precise bound on ϵ.

4. Two crucial lemmas. In this section we shall prove two region-growing lemmas that will be central to our multicut algorithm. We shall prove these in sufficient generality so that they can be applied to the other versions of the multicommodity flow problem as well.

Given a graph $G = (V, E)$, a capacity function $c : E \rightarrow \Re^+$ and distance labels $d : E \rightarrow \Re^+$ on the edges, define $B = \sum_{e \in E} d_e c_e$. A subset of vertices $V' \subseteq V$ is provided to the region-growing algorithm as the set of candidate roots from which regions will be grown. In our case, V' is the set of sources and sinks. We associate a variable y_S with each subset $S \subset V$; initially $y_S = 0$ for all S. The cut associated with a set S, denoted by $\nabla(S)$, is the set of edges with exactly one end point in S. The capacity of the cut, $C_{\nabla(S)}$, is $\sum_{e \in \nabla(S)} c_e$.

4.1. Growing a region. A region is grown in a radial manner starting from a root vertex, r. The order in which vertices are included in the region is the same as the order in which Dijkstra's algorithm finds shortest paths to vertices from r. We begin by picking a vertex, $r \in V'$, and assign it a weight $wt(r) = B/q$, where $q = |V'|$.

At any point in the algorithm we identify a set, A, as the active set and raise its variable y_A. Initially, the active set is $\{r\}$. Define the weight enclosed by the set A as

$$wt(A) = \sum_{S \subseteq A} y_S C_{\nabla(S)} + wt(r).$$

It is important that the y_S's must form a packing, i.e., $\forall e \in E$: $\sum_{S: e \in \nabla(S)} y_S \leq d_e$. Thus, if while raising y_A we find that $\sum_{e \in \nabla(S)} y_S = d_e$ for some edge $e = (u, v) \in \nabla(A)$, $u \in A$, we make the set $A \cup \{v\}$ active, i.e., $A \leftarrow A \cup \{v\}$, and start increasing the variable corresponding to it. We keep growing the active set in this manner, one vertex at a time, until

$$C_{\nabla(A)} \leq \epsilon \cdot wt(A) \tag{1}$$

is satisfied, where ϵ is a constant that will be set appropriately while applying the lemma. Let $\mathcal{R}$ denote the active set for which condition 1 is satisfied.

Define the *radius* of A, $rad(A) = \sum_{S \subseteq A} y_S$.

LEMMA 4.1. $rad(\mathcal{R}) < \ln(q + 1)/\epsilon$.

Proof. The claim is trivial if $rad(\mathcal{R}) = 0$, so assume $rad(\mathcal{R}) > 0$. Let $S_1, S_2, \ldots, S_l$ denote the successive sets for which the variable $y_S > 0$. It is easy to check that these sets are nested, i.e., if $i < j$ then $S_i \subset S_j$. In what follows we denote the value of the variable y_{S_i} by y_i and $C_{\nabla(S_i)}$ by C_i.

Since, while raising the variable y_{S_i} from 0 to y_i, condition 1 was not satisfied

$$C_i \geq \epsilon \cdot wt(S_i).$$

From our definition of the weight enclosed by a set, it follows that

$$\begin{aligned} wt(S_i) &= wt(S_{i-1}) + y_i C_i \\ &\geq wt(S_{i-1}) + \epsilon y_i wt(S_i), \end{aligned} \tag{2}$$

where for $i = 1$ we let $wt(S_0) = wt(r)$ in the above equation. Note that $wt(S_{i-1}) > 0$ for all i and hence $0 < \epsilon y_i < 1$. Thus,

$$wt(S_i) \geq \frac{wt(S_{i-1})}{(1 - \epsilon y_i)}.$$

Hence,

$$wt(S_l) \geq \frac{wt(r)}{(1-\epsilon y_1)(1-\epsilon y_2)\cdots(1-\epsilon y_l)}$$

$$= \frac{B}{q(1-\epsilon y_1)(1-\epsilon y_2)\cdots(1-\epsilon y_l)}.$$

Since the y_S's form a packing ($\forall e \in E : \sum_{S:e\in\nabla(S)} y_S \leq d_e$), it follows that

$$\sum_{i=1}^{l} y_i C_i = \sum_{i=1}^{l}\left(y_i \sum_{e\in\nabla(S_i)} c_e\right) = \sum_{e\in E}\left(c_e \sum_{S:e\in\nabla(S)} y_S\right) \leq \sum_{e\in E} c_e d_e = B,$$

and hence,

$$wt(S_l) = \sum_{i=1}^{l} y_i C_i + wt(r) \leq B + \frac{B}{q} = B\left(1+\frac{1}{q}\right).$$

Therefore,

$$\frac{B}{q(1-\epsilon y_1)(1-\epsilon y_2)\cdots(1-\epsilon y_l)} \leq wt(S_l) \leq B\left(1+\frac{1}{q}\right),$$

which implies that

$$\frac{1}{(1-\epsilon y_1)(1-\epsilon y_2)\cdots(1-\epsilon y_l)} \leq q+1.$$

Taking natural logs we get

$$\sum_{i=1}^{l} \ln(1-\epsilon y_i)^{-1} \leq \ln(q+1).$$

From (2) it follows that $0 < \epsilon y_i < 1$, $1 \leq i \leq l$. Since $\ln(1-x)^{-1} > x$ for $0 < x < 1$,

$$\epsilon \sum_{i=1}^{l} y_i < \ln(q+1).$$

Thus, $rad(\mathcal{R}) = \sum_{i=1}^{l} y_i < \ln(q+1)/\epsilon$. □

Let $dist_d(u,v)$ denote the shortest path distance between u and v under the distance label assignment d. Consider vertex $v \in \mathcal{R}$. Let S_i be the first set containing v, i.e., $v \in S_i - S_{i-1}$. Then from the manner in which we grow the region it follows that

$$dist_d(r,v) = rad(S_{i-1}). \tag{3}$$

COROLLARY 4.2. *For all $u,v \in \mathcal{R}$, $dist_d(r,u) \leq rad(\mathcal{R})$ and $dist_d(u,v) \leq 2rad(\mathcal{R})$.*

4.2. Growing disjoint regions. Having grown a region rooted at an arbitrary vertex in V', remove all vertices contained in the region and grow another region starting from a new root picked from V'. Continue in this manner until the residual graph contains no vertex of V'. Let $\mathcal{R}_1, \mathcal{R}_2 \dots, \mathcal{R}_p$ denote the regions formed. It is easy to see that these are disjoint. Clearly, $p \leq |V'| = q$.

Let $M = \nabla(\mathcal{R}_1) \cup \nabla(\mathcal{R}_2) \cup \cdots \cup \nabla(\mathcal{R}_p)$.

LEMMA 4.3. $\sum_{e \in M} c_e \leq 2\epsilon B$.

Proof. Let $G_i = (V_i, E_i)$ be the graph obtained by deleting vertices contained in $\cup_{j=1}^{i-1} \mathcal{R}_j$ ($G_1 = G$). Furthermore, let $C^i_{\nabla(S)}$ denote the capacity of the cut $\nabla(S)$ in G_i. Note that $\sum_{e \in M} c_e = \sum_{i=1}^p C^i_{\nabla(\mathcal{R}_i)}$.

Each region $\mathcal{R}_i$ satisfies condition 1. Therefore, $C^i_{\nabla(\mathcal{R}_i)} \leq \epsilon . wt(\mathcal{R}_i)$, $1 \leq i \leq p$. Hence,

$$\sum_{i=1}^p C^i_{\nabla(\mathcal{R}_i)} \leq \epsilon \sum_{i=1}^p wt(\mathcal{R}_i)$$
$$= \epsilon \left(\sum_{i=1}^p \sum_{S \subseteq \mathcal{R}_i} y_S C^i_{\nabla(S)} + \sum_{i=1}^p wt(r_i) \right),$$

where r_i is the root of region $\mathcal{R}_i$. Since the y_S's form a packing,

$$\sum_{i=1}^p \sum_{S \subseteq \mathcal{R}_i} y_S C^i_{\nabla(S)} \leq \sum_{e \in E} d_e c_e = B.$$

Also,

$$\sum_{i=1}^p wt(r_i) = \frac{B}{q} p \leq B.$$

Thus, $\sum_{i=1}^p C^i_{\nabla(\mathcal{R}_i)} \leq 2\epsilon B$, and hence $\sum_{e \in M} c_e \leq 2\epsilon B$. □

When growing region $\mathcal{R}_j$ in the graph G_j, we have that $\sum_{e \in E_j} d_e c_e \leq \sum_{e \in E} d_e c_e = B$. However, the proof of Lemma 4.1 goes through without any modifications. Regarding (3), note that $rad(S_{i-1})$ is now the shortest path distance between r and v in the graph G_j; the shortest distance between these vertices in G might be even smaller. Hence Corollary 4.2 still holds.

COROLLARY 4.4. $\sum_{i=1}^p C_{\nabla(\mathcal{R}_i)} \leq 4\epsilon B$.

Proof. An edge in M occurs in at most two cuts $\nabla(\mathcal{R}_i)$, $1 \leq i \leq p$. Thus,

$$\sum_{i=1}^p C_{\nabla(\mathcal{R}_i)} \leq 2 \sum_{e \in M} c_e \leq 4\epsilon B. \qquad \square$$

The time complexity of growing disjoint regions is $O(m + n \log n)$ as our algorithm is essentially the same as Dijkstra's algorithm for shortest paths.

5. Approximate max-flow min-multicut theorem. Clearly, the max flow, F, is less than the weight of the minimum multicut, M, i.e., $F \leq M$.

The main result of this section is an algorithm that finds a multicut of weight at most $F \cdot O(\log k)$. We state this as a theorem for later reference.

THEOREM 5.1 (approximating the minimum multicut). *Consider an instance of the* MULTICUT *problem specified by a graph* $G = (V, E)$*, a capacity function* $c : E \rightarrow \Re^+$*, and* k *pairs of vertices. One can, in polynomial time, find a multicut separating the specified pairs of vertices having weight within a factor* $O(\log k)$ *of the maximum flow over these pairs.*

Since M is the minimum multicut, $M \leq F \cdot O(\log k)$. Thus the ratio of the optimal integral solution to the optimal fractional solution of the dual program is at most $O(\log k)$.

COROLLARY 5.2 (approximate max-flow min-multicut theorem). $F \leq M \leq F \cdot O(\log k)$.

Furthermore, this bound on the ratio of the minimum multicut and maxflow is tight, as shown in Theorem 5.4.

For planar graphs, Tardos and Vazirani [TV] obtain a constant factor approximation for the minimum multicut. Garg, Vazirani, and Yannakakis [GVY] approximate the minimum multicut on trees to within twice the optimal. They also give a factor-$\frac{1}{2}$ approximation algorithm for maximum integral multicommodity flow on trees, and they show that even for planar graphs the integrality gap for flow is unbounded, thereby ruling out LP duality based methods for approximating maximum integral multicommodity flow. Both these results also establish approximate max-flow min-multicut theorems.

5.1. Finding the multicut. First, solve the dual LP to obtain a set of distance labels, d_e, $e \in E$. Next, grow regions as in §4. The constant ϵ and the set V' are chosen to ensure that $\nabla(\mathcal{R}_1) \cup \nabla(\mathcal{R}_2) \cup \cdots \cup \nabla(\mathcal{R}_p)$ is a multicut.

The vertices of V that are the source for some commodity form the set V', i.e., $V' = \cup_{i=1}^{k}\{s_i\}$. Thus, $s_i \in \cup_{j=1}^{p}\mathcal{R}_j$, $1 \leq i \leq k$. Now if we can choose ϵ so that no two vertices in $\mathcal{R}_i$, $1 \leq i \leq p$, share an index, we shall be finished.

LEMMA 5.3. *If* $\epsilon = 2\ln(k+1)$*, then no two vertices in* $\mathcal{R}_i$ *share an index.*

Proof. Note that $q = |V'| \leq k$. Therefore, if $\epsilon = 2\ln(k+1)$, then, by Lemma 4.1, $rad(\mathcal{R}_i) < \frac{1}{2}$. Hence, by Corollary 4.2, the distance between any two vertices in $\mathcal{R}_i$ is less than 1. Since the assignment of distance labels to edges is such that the distance between any two vertices sharing an index is at least 1, no two vertices in $\mathcal{R}_i$ share an index. □

Substituting this choice of ϵ into Lemma 4.3, we find that the multicut obtained has weight at most $4B\ln(k+1)$. Since $B = \sum_{e \in E} d_e c_e = F$, the max flow, the weight of the multicut is within a factor $4\ln(k+1)$ of the max flow.

5.2. A tight example.

THEOREM 5.4. *For all* $k, n, k \leq n$*, there exists an* n *vertex graph,* G*, and* $\Omega(k^2)$ *pairs of vertices in* G *such that the ratio between the minimum multicut and the max flow is* $\Omega(\log k)$.

Proof. As in [LR], we use an expander graph and similar arguments, but here we need to choose an appropriate set of source–sink pairs and consider cuts into many parts. Let $G = (V, E)$ be a k-vertex, bounded degree expander graph (each vertex has degree at most d, for an appropriate constant d). Every vertex has at most $\frac{k}{2}$ vertices within distance $\log_d\left(\frac{k}{2}\right)$. Thus, G has $\Omega(k^2)$ pairs of vertices that are a distance $\log_d\left(\frac{k}{2}\right)$ or more apart. Let these be the pairs for the MULTICUT instance. All edges of the graph have unit capacity, and hence the total capacity of the edges is $O(k)$. Since each flow path is $\Omega(\log k)$ long, the maximum flow is $O(k/\log k)$.

The optimum multicut induces a partition of the vertex set of G. Since G is an expander, any set S in the partition has $\Omega(|S|)$ edges running across it, provided $|S| \leq \frac{k}{2}$. Any subgraph with more than $\frac{k}{2}$ vertices has pairs that are $\log_d\left(\frac{k}{2}\right)$ apart, and hence contains a pair of vertices that share an index. Thus, no set in the partition induced by the multicut has more than $\frac{k}{2}$ vertices, and so each set S in the partition has $\Omega(|S|)$ edges running across it. Hence the number of edges in the multicut is $\Omega(k)$. This yields a ratio of $\Omega(\log k)$ between the weight of the minimum multicut and the maxflow.

Note that splitting an edge by adding vertices on the edge does not change the max flow or the minimum multicut. We modify G into an n-vertex graph by adding an appropriate number of vertices on the edges of G. □

We remark that, in the case of the multiway cut problem [DJPSY] (i.e., the special case of the multicut problem where the given set of source–sink pairs for the commodities consists of all pairs of vertices from a given subset S of terminals), the gap between min cut and max flow is much smaller; it is at most $2 - \frac{2}{k}$ [Cu]. Of course, if S is the whole set of vertices (i.e., there is one commodity for every pair of vertices), the problem is trivial and there is no gap: max flow = min cut = total capacity of the graph.

6. Multicommodity flow with specified demands. We next consider the case when along with the source and sink for commodity i, $1 \leq i \leq k$, we are also specified a demand, dem(i), for the commodity. A multicommodity flow is *feasible* if it meets the demand for each commodity.

As in §4 we define the cut associated with S, denoted by $\nabla(S)$, as the set of edges with exactly one end point in S. The capacity of the cut, $C_{\nabla(S)}$, is $\sum_{e\in\nabla(S)} c_e$. The set S separates commodity i if and only if exactly one of $\{s_i, t_i\}$ is in S. The demand across the cut, $D_{\nabla(S)}$, is the sum of the demands of all commodities separated by set S. Clearly, the following condition is necessary for the existence of a feasible multicommodity flow.

Cut condition. For all $S \subseteq V$, $C_{\nabla(S)} \geq D_{\nabla(S)}$.

However, this condition is not sufficient. Extensive work has been done on characterizing graphs with distinguished sources and sinks for which the cut condition is both necessary and sufficient. Again, no complete characterization is known.

Klein et al. view this problem as follows: they wish to find the minimum factor, u, by which the capacity of the edges should be raised to ensure a feasible flow. Equivalently, they wish to find the maximum factor, f, such that it is possible to route simultaneously an amount $f \cdot$ dem(i) of each commodity while satisfying the capacity constraints. At optimality $f = \frac{1}{u}$.

Let q_i^j denote a path in G from s_i to t_i and $q_i^j(e)$ be the characteristic function of this path, i.e., $q_i^j(e) = 1$ if $e \in q_i^j$, 0 otherwise. Then an LP formulation for this problem is

$$\begin{array}{lll} \text{maximize} & f & \\ \text{subject to} & \sum_{i,j} f_i^j q_i^j(e) \leq c_e & \forall e \in E, \\ & \sum_j f_i^j \geq f.dem(i), & 1 \leq i \leq k, \\ & f_i^j \geq 0 & \forall q_i^j, \end{array}$$

where f_i^j is the flow along the path q_i^j. Thus, multicommodity flow is feasible if and only if f is at least 1.

We can formulate the dual LP which now calls for an assignment of nonnegative distance labels, d_e, to edges $e \in E$ so as to minimize $\sum_{e \in E} d_e c_e$, subject to the constraint $\sum_{l=1}^{k} dist_d(s_l, t_l) dem(l) \geq 1$, where $dist_d(u,v)$ is the shortest path distance between u and v under this assignment of distance labels. At optimality, we have $\sum_{e \in E} d_e c_e = f$.

How does f relate to the structure of the graph? The cut condition motivates the following definition. Define the *sparsest cut* as the cut that minimizes the ratio $C_{\nabla(S)}/D_{\nabla(S)}$. Let

$$\alpha = \min_{S \subseteq V} \frac{C_{\nabla(S)}}{D_{\nabla(S)}}.$$

Clearly, $f \leq \alpha$. Klein et al. show that the "throughput" f is at least $\alpha / O(\log C \log D)$, where C is the sum of all capacities and D is the total demand. This was later improved to $\alpha / O(\log n \log D)$ by Tragoudas [Trag].

We improve this result by providing a tighter bound on f.

THEOREM 6.1.

$$\frac{\alpha}{O(\log k \log D)} \leq f \leq \alpha.$$

Thus, for a multicommodity flow to be feasible it is necessary that the sparsest cut ratio, α, be at least 1 *and sufficient that it be* $O(\log k \log D)$.

Plotkin and Tardos [PT] give a method of scaling demands so that the log D factor in Theorem 6.1 can be replaced by log k. Hence, they improve the bound in Theorem 6.1 to $O(\log^2 k)$.

For uniform multicommodity flow on bounded degree expanders, the ratio between f and α is $O(\log n)$ (n is the number of vertices) [LR]. We can, as in Theorem 5.4, modify this by adding new vertices on edges to get a graph on n vertices and k commodities ($k \leq \frac{n(n-1)}{2}$) for which $f = \alpha / O(\log k)$. It is an open problem to bridge the gap between the $O(\log^2 k)$ bound of [PT] and this $O(\log k)$ example.

For planar graphs, Klein, Plotkin, and Rao [KPR] show that multicommodity flow is feasible if the sparsest cut ratio, α, is logarithmic.

6.1. Proof of Theorem 6.1. First solve the dual LP to obtain a set of distance labels, d_e, $e \in E$. By LP duality we have that the optimal "throughput" f is equal to $B = \sum_{e \in E} d_e c_e$. We will find a cut whose ratio of capacity to demand is within a factor $\rho = 16 \ln(k+1) \log D$ of f (and hence also of the optimal ratio α).

As in Klein et al., the algorithm proceeds in phases. Each phase involves growing regions as in §4. If the source s_j and sink t_j of a commodity belong to the same region in a phase we will say that commodity j is routed during that phase. In each phase we only consider the commodities that have not been routed so far, i.e., the set V' of candidate roots consists of the vertices that are sources for some unrouted commodity. Thus, $q = |V'| \leq k$. Let the *residual demand* at phase i, D_i, be the total demand over commodities which have not been routed in phases 1 to $i-1$. We set the rate of expansion for phase i to $\epsilon_i = \rho D_i / 8$. The purpose is to route a significant fraction of the demand while keeping the cut small. (Contrast this with Lemma 5.3 where ϵ was chosen so that no commodity had its endpoints in the same region.)

We claim that at least one of the regions $\mathcal{R}_j$ in one of the phases has ratio of capacity to demand at most ρf. Suppose that this is not the case, i.e., $C_{\nabla(\mathcal{R}_j)}/D_{\nabla(\mathcal{R}_j)} > \rho f$

for all regions. We will derive a contradiction to the constraint $\sum_{l=1}^{k} dist_d(s_l, t_l)dem(l) \geq 1$.

Consider phase i. The residual demand after phase i is

$$D_{i+1} \leq \sum_{j=1}^{p} D_{\nabla(\mathcal{R}_j)} \leq \frac{1}{\rho f} \sum_{j=1}^{p} C_{\nabla(\mathcal{R}_j)}.$$

From Corollary 4.4 we get

$$D_{i+1} \leq \frac{4\epsilon_i B}{\rho f} = \frac{4\epsilon_i f}{\rho f} = \frac{D_i}{2}.$$

Since $q \leq k$ it follows from Lemma 4.1 and Corollary 4.2 that any two vertices in the same region are less than $2\ln(k+1)/\epsilon_i$ apart. Thus for any commodity l that is routed in phase i we have

$$dist_d(s_l, t_l) < \frac{2\ln(k+1)}{\epsilon_i} = \frac{16\ln(k+1)}{\rho D_i}.$$

Therefore, the sum of the quantities $dist_d(s_l, t_l)dem(l)$ over the commodities routed in phase i is less than $16\ln(k+1)/\rho$. Since the residual demand is halved in each phase, all the commodities are routed after at most $\log D$ phases. Therefore, the sum over all commodities is

$$\sum_{l=1}^{k} dist_d(s_l, t_l)dem(l) < \frac{16\ln(k+1)\log D}{\rho} = 1,$$

a contradiction.

7. Product multicommodity flow. In this section, we will use our region-growing lemmas to establish an improved bound for the product multicommodity flow problem (defined in the introduction). If all the vertex weights are unity, this also gives a cleaner proof for the uniform multicommodity flow problem by dispensing with discretization (the rest of the ideas remain essentially the same). Another case of special interest is when a subset of the vertices (say k in number) have unit weights and the rest have weights zero. We call this the *k-terminal uniform multicommodity flow problem.* It can be viewed as the analogue of the multiway cut problem in the demands case. For this case, the gap between the maximum throughput and the minimum ratio of a cut is $O(\log k)$.

As stated in the previous section, the example of Leighton and Rao can be easily adapted to show that the $O(\log k)$ gap is essentially tight for the k-terminal uniform multicommodity flow problem (up to a constant factor). It is interesting to note that, in both the minimum multicut problem and the sparsest cut problem, there is a $\log k$ discrepancy between the uniform case (where we have all pairs among a given set of terminals) and the general case (arbitrary set of pairs): in the multicut problem the integrality gap is $2 - \frac{1}{k}$ in the uniform case versus $O(\log k)$ in the nonuniform case. In the sparsest cut problem it is $O(\log k)$ versus $O(\log^2 k)$. The first three of these bounds are essentially tight. Only the last gap is not known to be tight; we do not have an example for the sparsest cut problem that takes advantage of nonuniformity.

In the product multicommodity flow problem each vertex v has a (nonnegative) weight $w(v)$, and there is a commodity for each unordered pair u, v of vertices with demand $w(u)w(v)$. Let

$$\alpha = \min_{S\subseteq V} \frac{C_{\nabla(S)}}{w(S)w(\overline{S})}.$$

The throughput f is clearly bounded from above by α.

The dual program again calls for an assignment of nonnegative distance labels, d_e, to the edges $e \in E$, so that $\sum_{e\in E} d_e c_e$ is minimized subject to the constraint that $\sum_{u,v\in V} w(u)w(v)\,dist_d(u,v) \geq 1$, where the sum extends over all unordered pairs of vertices. At optimality we have

$$\sum_{e\in E} d_e c_e = f.$$

THEOREM 7.1. *For the product multicommodity flow problem, the maximum throughput f and the minimum ratio α of a cut satisfy the inequalities*

$$\frac{\alpha}{O(\log k)} \leq f \leq \alpha,$$

where k is the number of vertices having nonzero weight. We can find in polynomial time a cut whose ratio is within a factor $O(\log k)$ of the minimum ratio.

A closely related quantity to the ratio of a cut $\nabla(S)$ is its *flux*:

$$\frac{C_{\nabla(S)}}{\min(w(S), w(\overline{S}))}.$$

Let

$$\beta = \min_{S\subseteq V} \frac{C_{\nabla(S)}}{\min(w(S), w(\overline{S}))}.$$

Recall that W denotes the sum of the weights of all the vertices. Since $\frac{W}{2} \leq \max(w(S),\, w(\overline{S})) \leq W$, it follows that $\frac{\alpha W}{2} \leq \beta \leq \alpha W$. Therefore, Theorem 7.1 implies that the minimum flux β lies between $\frac{fW}{2}$ and $fW \cdot O(\log k)$. Also, a cut approximates the minimum ratio α within a factor $O(\log k)$ if and only if it approximates the minimum flux β within a factor $O(\log k)$.

7.1. Proof of Theorem 7.1. We follow the structure of the Leighton–Rao proof for the uniform multicommodity flow problem, except that we shall use the lemmas of §4. Also, it is not necessary to readjust adaptively the expansion rate ϵ of the regions and restart the procedure. First solve the dual LP to obtain a set of distance labels d_e. By LP duality, we have that f is equal to $B = \sum_{e\in E} d_e c_e$. We shall find a cut whose flux is at most $\hat{\beta} = fW(4\ln(k+1)+1)$; thus its ratio of capacity to demand is within a factor $2(4\ln(k+1)+1)$ of the throughput f and hence also of the optimal ratio α.

We find a good cut in two stages. In the first stage we grow regions as in §4 with V' as the set of vertices with nonzero weight; $|V'| = k$. We choose $\epsilon = \hat{\beta}W/4f$. If one of the regions has flux at most $\hat{\beta}$ we are finished. Suppose this is not the case. We show then that one of the regions contains vertices of weight at least $\frac{W}{2}$. If every region $\mathcal{R}_i$ contains vertices of weight less than $\frac{W}{2}$, then its flux is

$$\frac{C_{\nabla(\mathcal{R}_i)}}{\min(w(\mathcal{R}_i), w(\overline{\mathcal{R}_i}))} = \frac{C_{\nabla(\mathcal{R}_i)}}{w(\mathcal{R}_i)} > \hat{\beta}.$$

Therefore,

$$\sum_{i=1}^{p} C_{\nabla(\mathcal{R}_i)} > \hat{\beta} \sum_{i=1}^{p} w(\mathcal{R}_i) = \hat{\beta} W.$$

However, by Corollary 4.4 we have

$$\sum_{i=1}^{p} C_{\nabla(\mathcal{R}_i)} \leq 4\epsilon B = 4\epsilon f = \hat{\beta} W.$$

We conclude that one of the regions, say $\mathcal{R}*$, has weight at least $\frac{W}{2}$. Let r be the root of region $\mathcal{R}*$.

In the second stage we reset all the variables y_S to zero and grow a region from root r as in §4. However, now we do not check for condition 1 and stop only when all vertices are included in the region. Let $\{r\} = S_1 \subset S_2 \subset \cdots \subset S_t$ denote the sets with $y_S > 0$. We will argue that one of these sets has flux at most $\hat{\beta}$. Assume that this is not the case; we shall derive a contradiction to the constraint $\sum_{u,v \in V} w(u)w(v)\mathit{dist}_d(u,v) \geq 1$.

First observe that $\mathit{dist}_d(u,v) \leq \mathit{dist}_d(r,u) + \mathit{dist}_d(r,v)$. Therefore,

$$\sum_{u,v \in V} w(u)w(v)\mathit{dist}_d(u,v) \leq W \sum_{v \in V} w(v)\mathit{dist}_d(r,v).$$

If S_i is the smallest set containing vertex v then $\mathit{dist}_d(r,v) = \sum_{S \subset S_i} y_S$. Let $S_l = \mathcal{R}$ be the first set in the chain that contains all the vertices of $\mathcal{R}*$. The set $\mathcal{R}$ may be a proper superset of $\mathcal{R}*$, because some nodes may have been deleted from the graph in the previous stage by the time we grew the region around r. In any case, however, $\mathit{rad}(\mathcal{R}) \leq \mathit{rad}(\mathcal{R}*)$. Since $|V'| = k$, by Lemma 4.1

$$\mathit{rad}(\mathcal{R}) < \frac{\ln(k+1)}{\epsilon} = \frac{4f\ln(k+1)}{W\hat{\beta}}.$$

The total weighted distance of all the vertices from r is

$$\sum_{v \in V} w(v)\mathit{dist}_d(r,v) = \sum_{S} y_S(W - w(S)) = \sum_{S \subset \mathcal{R}} y_S(W - w(S)) + \sum_{S \supseteq \mathcal{R}} y_S(W - w(S)).$$

Now,

$$\sum_{S \subset \mathcal{R}} y_S(W - w(S)) \leq W \sum_{S \subset \mathcal{R}} y_S \leq W \cdot \mathit{rad}(\mathcal{R}) < \frac{4f\ln(k+1)}{\hat{\beta}}.$$

Since $w(\mathcal{R}) \geq \frac{W}{2}$, all supersets of $\mathcal{R}$ have weight at least $\frac{W}{2}$. Hence, the flux for these sets if $C_{\nabla(S)}/(W - w(S)) \geq \hat{\beta}$ and thus

$$\sum_{S \supseteq \mathcal{R}} y_S(W - w(S)) \leq \frac{1}{\hat{\beta}} \sum_{S \supseteq \mathcal{R}} y_S C_{\nabla(S)} \leq \frac{1}{\hat{\beta}} \sum_{e \in E} d_e c_e = \frac{f}{\hat{\beta}}.$$

Therefore, $\sum_{v \in V} w(v)\mathit{dist}_d(r,v) < \frac{f}{\hat{\beta}}(4\ln(k+1) + 1) < \frac{1}{W}$, and hence

$$\sum_{u,v \in V} w(u)w(v)\mathit{dist}_d(u,v) < 1,$$

a contradiction.

8. Applications. Several graph problems can be viewed as edge deletion problems. We wish to find a minimum weight set of edges whose removal yields a graph with a desired structure π [Ya]. Klein et al. [KARR] propose a method for approximating such a problem when the property π can be specified as a 2CNF $\equiv$ formula so that deleting edges from the graph corresponds to deleting clauses in the formula. In particular, they show how to model the minimum edge deletion graph bipartization problem, i.e., deleting a minimum weight set of edges so that the resulting graph is bipartite. The 2CNF $\equiv$ deletion problem is defined as follows.

> A 2 CNF $\equiv$ formula, F, is a weighted set of clauses of the form $P \equiv Q$ where P, Q are literals. Find a minimum weight set of clauses the deletion of which makes the formula satisfiable.

Klein et al. showed that this problem can be reduced to the minimum multicut problem. Construct a graph $G(F)$ whose vertex set is the set of literals in F. For each clause of the kind $P \equiv Q$ include two edges (P, Q) and $(\overline{P}, \overline{Q})$ of capacity equal to the weight of the clause $P \equiv Q$.

LEMMA 8.1. *A 2CNF $\equiv$ formula, F, is satisfiable if and only if no connected component of the graph $G(F)$ contains both a literal and its complement.*

Proof. An edge (P, Q) in $G(F)$ implies that the literals P and Q take the same truth value. Thus, if a literal and its complement occur in the same connected component then the 2CNF $\equiv$ formula is not satisfiable.

Conversely, note that if two literals P, Q are in the same connected component then their complementary literals $\overline{P}$, $\overline{Q}$ are also in the same connected component. Thus, the components can be paired, so that in each pair one component contains a set of literals and the other contains the complementary literals. We can now obtain a satisfying assignment by setting, for each pair of components, the literals of one component to true (and the other's to false). □

Let M be a minimum weight set of edges whose removal separates the pairs of complementary literals in $G(F)$ and let W be the minimum weight set of clauses whose deletion makes F satisfiable. Then we have the following lemma.

LEMMA 8.2. $wt(W) \leq wt(M) \leq 2wt(W)$.

Proof. The minimum multicut, M, in $G(F)$ corresponds to a set of clauses (of weight at most $wt(M)$) whose deletion makes the formula satisfiable. Hence, $wt(W) \leq wt(M)$.

Each clause of F corresponds to two edges in $G(F)$. Thus the set W corresponds to a multicut in $G(F)$ of weight at most $2wt(W)$. Therefore, $wt(M) \leq 2wt(W)$. □

Finding the set of edges, M, is exactly the MULTICUT problem on the graph $G(F)$ with every pair of complementary literals forming a source–sink pair. Thus, the number of pairs, k, is equal to the number of variables in the formula, n, and hence by Theorem 5.1 we can approximate M to within a factor $O(\log n)$. Using Lemma 8.2 we get the following theorem.

THEOREM 8.3. *Given a 2CNF $\equiv$ formula, one can in polynomial time find a set of clauses of weight at most a factor $O(\log n)$ of the minimum weight set of clauses whose deletion makes the formula satisfiable.*

COROLLARY 8.4. *The edge-deletion graph bipartization problem can be approximated within a factor $O(\log n)$ in polynomial time.*

We leave open the question of whether these problems can be approximated within some constant factor. We know they are both MAX SNP-hard [PY] and hence do not have a polynomial-time approximation scheme unless $P = NP$ [ALMSS].

Acknowledgment. We wish to thank Phil Klein for simplifying the formulation in §6, which made the presentation more uniform.

REFERENCES

[ALMSS] S. Arora, C. Lund, R. Motwani, M. Sundan, and M. Szegedy, *Proof verification and hardness of approximation problems*, in Proc. 32nd IEEE Symposium on Foundations of Computer Science, IEEE Press, Piscataway, NJ, 1992, pp. 14–23.

[Aw] B. Awerbuch, *Complexity of network synchronization*, J. Assoc. Comput. Mach., 32 (1985), pp. 804–823.

[Cu] W. H. Cunningham, *The optimal multiterminal cut problem*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, 5 (1991), pp. 105–120.

[DJPSY] E. Dahlhaus, D. S. Johnson, C. H. Papadimitriou, P. D. Seymour, and M. Yannakakis, *The complexity of multiway cuts*, in Proc. 24th ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1992, pp. 241–251.

[EFS] P. Elias, A. Feinstein, and C. F. Shannon, *A note on the maximum flow through a network*, IRE Trans. Inform. Theory IT, 2 (1956), pp. 117–119.

[FF] L. R. Fort, Jr. and D. R. Fulkerson, *Maximal flow through a network*, Canad. J. Math., 8 (1956).

[GV] N. Garg and V. V. Vazirani, *A characterization of the vertices and edges of the $s-t$ cut polyhedron, with algorithmic applications*, in Proc. Integer Programming and Combinatorial Optimization, 1993.

[GVY] N. Garg, V. V. Vazirani, and M. Yannakakis, *Primal-dual approximation algorithms for integral flow and multicut in trees, with applications to matching and set cover*, in Proc. 20th International Collection on Automata, Languages, and Programming, Springer-Verlag, Berlin, New York, Heidelberg, 1992, pp. 64–75.

[Hu] T. C. Hu, *Multicommodity network flows*, Oper. Res., 11 (1963), pp. 344–360.

[KARR] P. Klein, A. Agrawal, R. Ravi, and S. Rao, *Approximation through multicommodity flow*, in Proc. 31st Symposium on Foundations of Computer Science, IEEE Press, Piscataway, NJ, 1990, pp. 726–737.

[KPR] P. Klein, S. Plotkin, and S. Rao, *Excluded minors, network decomposition, and multi-commodity flow*, in Proc. 25th ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1992, pp. 682–690.

[KST] P. Klein, C. Stein, and E. Tardos, *Leighton–Rao might be practical: faster approximation algorithms for concurrent flow with uniform capacities*, in Proc. 22nd ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1990, pp. 310–321.

[LR] F. T. Leighton and S. Rao, *An approximate max-flow min-cut theorem for uniform multicommodity flow problems with application to approximation algorithms*, in Proc. 29th IEEE Symposium on Foundations of Computer Science, IEEE Press, Piscataway, NJ, 1988, pp. 422–431.

[PY] C. H. Papadimitriou and M. Yannakakis, *Optimization, approximation and complexity classes*, J. Comput. System Sci., 43 (1991), pp. 425–440.

[PT] S. Plotkin and E. Tardos, *Improved bounds on the max-flow min-cut ratio for multicommodity flows*, in Proc. 25th ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1992, pp. 691–697.

[TV] E. Tardos and V. V. Vazirani, *Improved bounds for the max-flow min-multicut ratio for planar and $K_{r,r}$-free graphs*, Inform. Process. Lett., 47 (1993), pp. 77–80.

[Trag] S. Tragoudas, *Improved approximation for the minimum-cut ratio and the flux*, Math. Systems Theory, to appear.

[Ya] M. Yannakakis, *Edge-deletion problems*, SIAM J. Comput., 10 (1981), pp. 297–309.

[YKCP] M. Yannakakis, P. C. Kanellakis, S. C. Cosmadakis, and C. H. Papadimitriou, *Cutting and partitioning a graph after a fixed pattern*, in Proc. 10th International Collection on Automata, Languages, and Programming, Springer-Verlag, Berlin, New York, Heidelberg, 1983, pp. 712–722.

SIAM J. COMPUT.
Vol. 25, No. 2, pp. 252–271, April 1996

ROBUST CHARACTERIZATIONS OF POLYNOMIALS WITH APPLICATIONS TO PROGRAM TESTING*

RONITT RUBINFELD[†] AND MADHU SUDAN[‡]

Abstract. The study of self-testing and self-correcting programs leads to the search for robust characterizations of functions. Here the authors make this notion precise and show such a characterization for polynomials. From this characterization, the authors get the following applications. Simple and efficient self-testers for polynomial functions are constructed. The characterizations provide results in the area of coding theory by giving extremely fast and efficient error-detecting schemes for some well-known codes. This error-detection scheme plays a crucial role in subsequent results on the hardness of approximating some NP-optimization problems.

Key words. coding theory, program correctness, low-degree polynomial testing

AMS subject classifications. 68Q25, 68Q40, 68Q60

1. Introduction.

The study of program checkers [Blu88], [BK89], self-testing programs [BLR90], and self-correcting programs [BLR90], [Lip91] was introduced in order to allow one to use a program P to compute a function without trusting that P works correctly. A *program checker* checks that the program gives the correct answer on a particular input, a *self-testing program* for f tests that program P is correct on most inputs, and a *self-correcting program* for f takes a program P that is correct on most inputs and uses it to compute f correctly on every input with high probability. The program checker, self-tester, and self-corrector may call the program as a black box, are required to do something other than to actually compute the function, and should be much simpler and at least different from any program for the function f in the precise sense defined by [BK89]. It is straightforward to show that checkers, self-testers, and self-correctors for functions are related in the following way: If f has a self-tester and a self-corrector, then it can be shown that f has a program result checker. Conversely, if f has a checker, then it has a self-tester (though not necessarily a self-corrector). It is argued in [BK89] and [BLR90] that this provides an attractive alternative method for attacking the problem of program correctness.

One of the main goals of the research in the area of self-testing/correcting programs and program checking is to find general techniques for finding very simple and efficient self-testers, self-correctors, and checkers for large classes of problems. In fact, some success towards this goal has been achieved. For example, in [BK89], it is shown how to use techniques from the area of interactive proof systems in order to write checkers. Using these and other techniques, checkers (and hence self-testers) have been found for a variety of problems [AHK], [BK89], [Rub90], [Kan90], [BFLS91], [BF91]. If a function is random self-reducible, i.e., the value of the function at any

* Received by the editors September 23, 1993; accepted for publication (in revised form) August 9, 1994. This paper unifies and extends part of the results contained in Gemmell et al. [GLRSW91] and Rubinfeld and Sudan [RS92].

[†] Department of Computer Science, Cornell University, Ithaca, NY 14853 (ronitt@cs.cornell.edu). The research of this author was supported by Office of Naval Research Young Investigator grant N00014-93-1-0590, United States–Israel Binational Science Foundation grant 92-00226, DIMACS (the Center for Discrete Mathematics and Theoretical Computer Science, Princeton University), and National Science Foundation grant NSF-STC88-09648. Part of this work was done while this author was at Princeton University.

[‡] I.B.M. Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598 (madhu@watson.ibm.com). Part of this work was done while this author was a student at the University of California at Berkeley under the support of National Science Foundation Presidential Young Investigator grant CCR-8896202.

input can be inferred from its value at randomly chosen inputs, then it has a self-corrector [BLR90], [Lip91]. This provides self-correctors for a surprising range of functions, including the class of linear functions (homomorphisms between groups) and polynomials.

In the direction of characterizing functions that have self-testers, some success has been achieved in [BLR90]. They give a number of methods of constructing self-testers for functions, some of which we mention here. They observe that any checker for a function can be used to construct a self-tester for the function. They present a particular method of constructing self-testers for a variety of functions based on a method of bootstrapping from tests over smaller domains. They also show another method of constructing self-testers for all linear functions, i.e., functions that act as homomorphisms between groups, in other words satisfy $f(x) + f(y) = f(x + y)$ for a group operation $+$.

The main focus of this paper is to study and understand the functions which have self-testers and to broaden the class of functions that are known to have self-testers. The linearity tester of [BLR90] is the starting point for this paper. A particularly interesting feature of this linearity tester is that it breaks the task of self-testing a function into the two tasks of (1) testing it for certain "structural properties" and (2) using the structural property to then identify the function precisely. In this paper, we introduce a new notion—a function-family tester—which helps delineate these two tasks more clearly. We first introduce some terminology.

We work with functions defined over some finite domain $\mathcal{D}$. The *distance* between two functions f and g over the domain $\mathcal{D}$ is the fraction of points $x \in \mathcal{D}$ where the two functions disagree:

$$d(f,g) \equiv \frac{|\{x \in \mathcal{D} | f(x) \neq g(x)\}|}{|\mathcal{D}|}.$$

We say that two functions are *ϵ-close* if $d(f,g) \leq \epsilon$. In some of the informal discussions that follow, we drop the ϵ and just describe two functions as being close. In such cases, it is implied that we are talking of some small enough ϵ. In terms of this notion, a self-tester for a function f may be defined as follows.

An ϵ-self-tester T for a function f over a domain $\mathcal{D}$ is a (randomized) oracle program that takes as input a program P and behaves as follows:

- It accepts P if $d(P,f) = 0$.
- It rejects P (with high probability) if P and f are not ϵ-close.
- It behaves arbitrarily otherwise.

1.1. Testers for function families using robust characterizations. Let $\mathcal{F}$ be a family of functions. An *ϵ-function-family tester* T for the family $\mathcal{F}$ takes as input a program P and tests if there exists a function $f \in \mathcal{F}$ such that P is ϵ-close to f.

The notion of a function-family tester captures the notion of verifying properties of a function as follows: Let $\mathcal{P}$ be a property we wish to test for. Let $\mathcal{F}$ be the family of all functions that have the property $\mathcal{P}$. Then a function-family tester for $\mathcal{F}$ can be used to test if a program P "essentially" has the property $\mathcal{P}$ (i.e., there exists a function with property $\mathcal{P}$ that is close to P). To make some of these abstract definitions concrete, let us work with the simple example of the property of linearity among functions from $\mathcal{Z}_p$ to $\mathcal{Z}_p$. For this example, the family of functions we work with is $\mathcal{F}_{\text{linear}} \equiv \{f_a | a \in \mathcal{Z}_p, f_a(x) = a \cdot x\}$. Thus a tester for the family of linear functions verifies that the computation of a program P is essentially linear.

The existence of a function-family tester for any class of functions implies a powerful characterization of the family. In particular, consider any program that is rejected by the tester. In order to reject the program, the tester will have found some evidence in the small set of sampled points which "proves" that P can not be a member of $\mathcal{F}$. In other words, all members of $\mathcal{F}$ must satisfy some property on the set of inputs that are examined by the family tester. Thus all members of $\mathcal{F}$ satisfy a "local" property (by local we mean a property on a set of small size—we define this notion more formally in §2). Moreover, if all such local properties are satisfied, then the tester accepts the function, implying that these local constraints form a characterization of the family. Thus in order for a function family to have a tester, it needs to have a local characterization. In our example, such a local characterization of linear functions is the property that $\forall x, y \in \mathcal{Z}_p$, $f(x) + f(y) = f(x+y)$. If a function is not linear then there exists a counterexample of size three that proves that it is not linear.

However, local characterizations do not form a sufficient condition for the construction of testers. Typically an exact local characterization of a family of functions involves a universal quantification, which is not feasible to verify. In our example, the characterization of linear functions by the property $\forall x, y \in \mathcal{Z}_p$, $f(x)+f(y) = f(x+y)$ is not useful to test a purported linear function, since we cannot hope to efficiently test that this holds for all possible pairs x and y. Thus for a characterization to be useful for testing, it needs to be "robust," involving the words "for most" rather than "for all." Specifically, let $\mathcal{F}$ be the function family that satisfies the properties at all inputs and let f be any function that satisfies the properties at most inputs. Then f must be close to some $g \in \mathcal{F}$ (see §2 for a more formal definition). In our example, if $f(x) + f(y) = f(x + y)$ is satisfied by f for most x, y, then $f(x) = c \cdot x$ for most x and some constant c.

1.2. Our results on function family testing. One of the main emphases of this paper is to find robust characterizations for the family of low-degree univariate and multivariate polynomials. In §3, we start by describing some (well-known) local characterizations of univariate and multivariate polynomials and then prove that some of these characterizations are actually robust characterizations. As an immediate consequence, we get function-family testers for all low-degree polynomials over finite fields. For the case of polynomials over Z_p, our testers are very simple and do not even need to multiply elements of the field. Our testers are the first testers that directly attempt to test the total degree of a polynomial (as opposed to the testers of [BFLS91], [FGLSS91], and [AS92], all of which test that the degree in each variable is not too large). The proof of correctness of our tester also is different from the proofs of correctness of the other testers in that it does not rely on an inductive argument based on the number of variables. This allows for its "efficiency" to be independent of the number of variables and provides the hope for the existence of a tester with nearly optimal efficiency.

A second emphasis of this paper is the notion of test sets that allows us to use the results on function-family testing to obtain self-testers for specific functions. Informally, a *test set* is a set of points from the domain such that no two functions from the family $\mathcal{F}$ agree with each other on all the points from the test set. Our self-tester for a specific function f would require, as a description of f, its value on all points in a test set. The complexity (running time) of the self-tester will depend on the size of the test set.

1.3. Other implications of low-degree testing. The task of constructing family testers for the family of low-degree polynomials is closely related to the task of error detection in Reed–Solomon codes. In fact, a low-degree test can be described as a "randomized" error detector that determines whether the number of errors in a received word is small or not. In this sense, the error detectors we construct have the feature that they are highly efficient and can be used to get estimates on the distance of a received word from a valid codeword. This perspective can similarly be applied to the results of [BLR90] to get randomized error-detecting and correcting schemes for the Hadamard codes that probe the received word in only a constant number of bits to detect an error or find any bit of the codeword closest to the received word. In fact, it has been observed by M. Naor [Nao92] that these results can be used to construct codes for which error-detection/correction can be performed by uniform quasipolynomial-sized circuits of constant depth. In §7, we define the notion of a "locally testable code"—a notion that precisely describes the relationship between testing and error-correcting codes. We also provide applications of our testers to the construction of "locally testable codes" in the section.

A different perspective on the construction of family testers is to view it as the following approximation problem:

> Given a family of functions $\mathcal{F}$ and a function P, estimate the distance $d(P, \mathcal{F})$ between P and $\mathcal{F}$ to within a small multiplicative error.

A tester for a function family $\mathcal{F}$ essentially yields such an approximator (provided $d(P, \mathcal{F})$ is smaller than half) by defining some new quantities $\delta(P, \mathcal{F})$ that are easy to estimate by random sampling and then showing that some approximate relations hold between $\delta(P, \mathcal{F})$ and $d(P, \mathcal{F})$. For example, the linearity test of [BLR90] may be viewed as trying to approximate the distance $d(f, \mathcal{F}_{\text{linear}})$. To approximate this distance, they define the quantity $\delta(f, \mathcal{F}_{\text{linear}}) \equiv \Pr[f(x) + f(y) \neq f(x+y)]$ which is easy to approximate. Then they show that $\delta(f, \mathcal{F}_{\text{linear}})/3 \leq d(f, \mathcal{F}_{\text{linear}}) \leq 9/2\delta(f, \mathcal{F}_{\text{linear}})$. The testers given here define similar quantities related to low-degree polynomials and show similar approximate relationships. Such inequalities may be of independent interest.

The task of low-degree testing forms a central ingredient in the proof of MIP = NEXPTIME due to [BFL91]. The tester given here provides an alternate mechanism that works in their setting. The efficiency of low-degree testing also becomes very important to the ensuing results on hardness of approximations [FGLSS91], [ALMSS92], and therefore a lot of attention has been paid to this problem [BFL91], [BFLS91], [FGLSS91], [AS92]. However, all these results focus on tests that are close variants of the test given in [BFL91]. The low-degree test given here is fundamentally different from the ones mentioned above and originated from independent considerations in the work of [GLRSW91]. The efficiency of the tester shown here may also be found in [RS92]. It turns out that this tester is particularly well-suited to such multiple prover applications and provides a one round, constant prover proof that a function is a low degree polynomial over finite fields. This is observed in subsequent work of [ALMSS92] (see also [Sud92]) and follows by using an improved analysis for Lemma 5.3 from [AS92]. This turns out to play a crucial role in the NP = PCP$(\log n, O(1))$ result of [ALMSS92], which in turn provides hardness results for a wide variety of approximation problems. An exact description of the relevance of the various testers and the chronology of contributions maybe found in §8.

1.4. Organization of the paper. The rest of this paper is organized as follows. In §2 we formally define the notions of local characterizations—exact and robust. Section 3 lists some (well-known) exact characterizations of low-degree polynomials. Sections 4 and 5 show that two of these exact characterizations are robust. In §6, we describe the applications of these characterizations to self-testing of programs. In §7, we define a notion of locally testable codes (based on the notion of probabilistically checkable proofs) and show applications of our testers to such codes. Section 8 contains some concluding remarks.

2. Local characterizations: Exact and robust. In this section, we make precise the notion of a local characterization and what we mean by exact and robust characterizations. We will also isolate a parameter associated with the robust characterizations that captures the efficiency of the tester suggested by the characterization.

We will use $\mathcal{D}$ to represent a finite domain. We will consider here families of functions $\mathcal{F}$ where $f \in \mathcal{F}$ maps elements from $\mathcal{D}$ to a range $\mathcal{R}$. We illustrate these definitions using the example of linear functions. Here the domain and range are $\mathcal{Z}_p$ and the family of functions is $\{f_a | a \in \mathcal{Z}_p$ where $f_a(x) = a \cdot x\}$.

DEFINITION 2.1 (neighborhoods). *A* k-local neighborhood N *is an ordered tuple of (not necessarily distinct) k points from $\mathcal{D}$. A* k-local collection of neighborhoods $\mathcal{N}$ *is a set of k-local neighborhoods.*

DEFINITION 2.2 (properties). *A* k-local property $\mathcal{P}$ *is a function from* $(\mathcal{D} \times \mathcal{R})^k$ *to* $\{0, 1\}$. *We say that a function f satisfies a property $\mathcal{P}$ over a neighborhood N if* $\mathcal{P}(\{(x, f(x))\}_{x \in N}) = 1$.

DEFINITION 2.3 (exact characterizations). *A property $\mathcal{P}$ over a collection of neighborhoods $\mathcal{N}$ is an* exact characterization *of a family of functions $\mathcal{F}$ if a function f satisfies $\mathcal{P}$ over all neighborhoods $N \in \mathcal{N}$ exactly when $f \in \mathcal{F}$. The characterization is k-local if the property $\mathcal{P}$ (and the collection $\mathcal{N}$) is k-local.*

In our example, the collection of neighborhoods $\mathcal{N} = \{(x, y, x+y) | x, y \in \mathcal{Z}_p\}$. The property $\mathcal{P}$ is 3-local and is satisfied by f on the triple (x_1, x_2, x_3) if $f(x_1) + f(x_2) = f(x_3)$. Thus over the collection of neighborhoods $\mathcal{N}$, $\mathcal{P}$ gives a 3-local characterization of the family of linear functions.

DEFINITION 2.4 (robust characterizations). *A property $\mathcal{P}$ over a collection of neighborhoods $\mathcal{N}$ is said to be an* (ϵ, δ)-robust characterization *of $\mathcal{F}$, if whenever a function f satisfies $\mathcal{P}$ on all but a δ fraction of the neighborhoods in $\mathcal{N}$, it is ϵ-close to some function $g \in \mathcal{F}$. Moreover, all members of $\mathcal{F}$ satisfy $\mathcal{P}$ on all neighborhoods in $\mathcal{N}$.*

To continue with the example of linear functions, the theorem of [BLR90] can be used to say that $\mathcal{P}$ over the neighborhood $\mathcal{N}$ is a $(\frac{9}{2}(\frac{2}{9} - \alpha), \frac{2}{9} - \alpha)$-robust characterization of linear functions for any constant α.

The exact constant ϵ determining closeness is not very important for the family of multivariate polynomials. For most of the characterizations we consider here, it can be shown that any function f is $((1 + o(1))\delta)$-close to some member g of $\mathcal{F}$ if f is $\frac{1}{4}$-close to g and violates only a δ fraction of the neighborhood constraints. Thus for the purposes of this paper, we fix the value of ϵ to be $\frac{1}{4}$.

In order to test if f is close to some member of $\mathcal{F}$, one would need to sample at least $\frac{1}{\delta}$ of the neighborhoods in $\mathcal{N}$ and test if $\mathcal{P}$ holds on these neighborhoods. Hence, the parameter $\frac{1}{\delta}$ is referred to as the *efficiency* of the characterization.

3. Exact characterizations of polynomials. In this section, we start by describing some (well-known) exact local characterizations of polynomial functions. In later sections, we will show that some of these characterizations can be made robust.

The family of degree-d polynomials can be characterized in a number of ways. The different characterizations arise from looking at different collections of neighborhoods $\mathcal{N}$. The property P has to remain invariant in the following sense: P will be satisfied by f on a neighborhood N if there exists a polynomial that agrees with f on all points in N. The complexity of a neighborhood test, i.e., testing whether a constraint is being satisfied by a neighborhood, is also influenced by the choice of the neighborhood. Thus by choosing the characterizations appropriately, we might be able to tradeoff the simplicity of the neighborhood test against the number of times the test needs to be repeated. The different characterizations also have to be qualified by different restrictions on the underlying ring. For instance, some characterizations hold only for finite fields while others hold only for rings of the form $\mathcal{Z}_m$. We will take care to point out the restrictions on the characterizations. We give examples of possible neighborhoods and their corresponding tests.

1. *Univariate polynomials.* The following characterization of univariate polynomials holds for a function f mapping a ring R to itself.
   (a) *Characterization.* $f : R \mapsto R$ is a polynomial of degree at most d if and only if $\forall x_0, \ldots, x_{d+1} \in R$, there exists a polynomial $g_{x_0,\ldots,x_{d+1}}$ of degree at most d such that $f(x_i) = g_{x_0,\ldots,x_{d+1}}(x_i)$.
   (b) *Neighborhood structure.* $\mathcal{N} = R^{d+2}$, i.e., all possible (multi)subsets of R of size $d+2$.
   (c) *Complexity of neighborhood test.* A test of the above nature involves finding the (unique) degree d polynomial g that agrees with f at the points $x_0, \ldots, x_d$ and then evaluating $g(x_{d+1})$ and verifying that this equals $f(x_{d+1})$. Standard interpolation techniques (see, for instance, [dW70]) imply that this is equivalent to computing coefficients $\alpha_0, \ldots, \alpha_{d+1}$, where the $\{\alpha_i\}$'s depend only on the $\{x_i\}$'s, and verifying that $\sum_{i=0}^{d+1} \alpha_i \cdot f(x_i) = 0$. The α_i's can be computed using elementary algorithms with $O(d^2)$ additions, subtractions, and multiplications over R.
2. *Univariate polynomials using evenly spaced points.* This characterization works over the ring $\mathcal{Z}_m$. Let $\alpha_i = \binom{d+1}{i}(-1)^{i+1}$. The interpolation identity for degree d polynomials on evenly spaced points, $x, x+h, \ldots, x+(d+1)\cdot h$, reduces to $\sum_{i=0}^{d+1} \alpha_i f(x + i \cdot h) = 0$. We refer to x as the *starting point* and h as the *offset.*
   (a) *Characterization.* $f : \mathcal{Z}_m \mapsto \mathcal{Z}_m$ is a polynomial of degree at most d if and only if $\forall x, h \in \mathcal{Z}_m$, $\sum_{i=0}^{d+1} \alpha_i f(x + i \cdot h) = 0$.
   (b) *Neighborhood structure.* Define the neighborhood sets $N_{x,h} \equiv \{x + i \cdot h\}_{i=0}^{d+1}$. Then the neighborhood collection is $\mathcal{N} = \bigcup_{x,h \in \mathcal{Z}_m} N_{x,h}$.
   (c) *Complexity of neighborhood test.* Notice that the constants α_i are now independent of x and h and can be precomputed once and for all. In fact, due to the special relationship between the α_i's, given the value of f at the points $x + i \cdot h$, we can compute the above summation with $O(d^2)$ additions and subtractions and *no multiplications* (see appendix).
3. *Multivariate polynomials using lines.* This characterization applies to m-variate functions over a finite field F. Define the notion of a *line* through the space F^m as follows: For $\hat{x}, \hat{h} \in F^m$, the line $l_{\hat{x},\hat{h}}$ through $\hat{x}$ with offset $\hat{h}$ is the set of points $\{\hat{x} + i \cdot \hat{h} | i \in F\}$. We will often refer to the line in its parametric form $l_{\hat{x},\hat{h}}(i)$. Observe that a polynomial f of total degree d, restricted to a line $l_{\hat{x},\hat{h}}(i)$ becomes a univariate polynomial of degree at most d in the parameter i. This gives us the following characterization of degree d polynomials over sufficiently large finite fields ($|F| \geq 2d+1$).[1]
   (a) *Characterization.* The function $f : F^m \mapsto F$ is a polynomial of degree at most d if and only if $\forall \hat{x}, \hat{h} \in F^m$, f restricted to $l_{\hat{x},\hat{h}}(i)$ is a univariate polynomial in i of degree at most d (see appendix for a proof).
   (b) *Neighborhood structure.* Let the neighborhoods be lines. Then $\mathcal{N} \equiv$

[1] The above characterization is not the tightest possible in its requirement of the parameter $|F|$. Indeed, for the case of fields of prime order this can be improved to the optimal case $|F| \geq d+2$ and this has been shown recently in [FS94]. For arbitrary finite fields, it turns out that $|F| \geq d+2$ is not a sufficient condition for this characterization to hold. A counterexample to this effect is also shown in [FS94].

$\{N_{\hat{x},\hat{h}} = l_{\hat{x},\hat{h}} | \hat{x}, \hat{h} \in F^m\}$.

   (c) *Complexity of neighborhood test.* In this form the characterization is not very local since the counterexamples are lines, i.e., collections of $|F|$ points. But this characterization is interesting to us because it says that the characterization of multivariate polynomials can be reduced to the characterization of univariate polynomials (on these lines). Thus we find that we can now use, for instance, characterization 1 to find counterexamples of size at most $d+2$. The complexity of a neighborhood test here is no more than the complexity of the neighborhood test in characterization 1.

4. *Multivariate polynomials using axis parallel lines.* This characterization is a specialization of the characterization above, in terms of special lines—*axis-parallel lines.* We say that a line is *axis parallel* if the offset $\hat{h}$ contains only one nonzero coordinate.
   (a) *Characterization.* $f : F^m \mapsto F$ is a polynomial of degree at most d in each variable if and only if $\forall$ axis-parallel lines, f restricted to the line is a univariate polynomial of degree at most d. Notice that here we characterize polynomials differently, i.e., in terms of individual degree in each variable rather than total degree.
   (b) *Neighborhood structure.* The neighborhoods here are sets of the form $N_{i,\hat{\beta}} \equiv \{(\beta_1, \ldots, \beta_{i-1}, t, \beta_i, \ldots, \beta_{m-1}) | t \in F\}$, for every choice of $\hat{\beta} \in F^{m-1}$ and every choice of $i \in \{1, \ldots, m\}$. Then

$$\mathcal{N} = \bigcup_{i \in \{1,\ldots,m\}, \hat{\beta} \in F^{m-1}} N_{i,\hat{\beta}}.$$

   (c) *Complexity of neighborhood test.* The complexity of a neighborhood test is the same as the complexity of characterization 1.

5. *Multivariate polynomials*: *Evenly spaced points.* A combination of characterizations 2 and 3 gives the following characterization of polynomials over $\mathcal{Z}_p$, provided p is large enough for characterization 3 to hold.
   (a) *Characterization.* $f : \mathcal{Z}_p^m \mapsto \mathcal{Z}_p$ is a polynomial of degree at most d if and only if $\forall \hat{x}, \hat{h} \in \mathcal{Z}_p^m$, $\sum_{i=0}^{d+1} \alpha_i f(\hat{x}+i\hat{h}) = 0$, where $\alpha_i = (-1)^{i+1}\binom{d+1}{i}$.
   (b) *Neighborhood structure.* The neighborhoods here are of the form $N_{\hat{x},\hat{h}} \equiv \{\hat{x} + i\hat{h} | i \in \{0, \ldots, d+1\}\}$. Then $\mathcal{N} \equiv \bigcup_{\hat{x},\hat{h} \in \mathcal{Z}_p^m} N_{\hat{x},\hat{h}}$.
   (c) *Complexity of neighborhood test.* The complexity of this neighborhood test is the same as the complexity in characterization 2.

6. *Multivariate polynomials*: *Evenly spaced points* 2. This characterization is a trivial consequence of the characterization above and seems weaker since its neighborhood structure is larger than those of the ones above. But it turns out that this characterization is much more useful due to the kind of robustness it yields. This characterization holds for polynomials over $\mathcal{Z}_p$, for $p > 10d$.
   (a) *Characterization.* $f : \mathcal{Z}_p^m \mapsto \mathcal{Z}_p$ is a polynomial of degree at most d if and only if $\forall \hat{x}, \hat{h} \in \mathcal{Z}_p^m$, the values of f at the points $\{\hat{x}+i\hat{h} | i \in \{0, \ldots, 10d\}\}$ agree with some univariate polynomial g of degree at most d in t.
   (b) *Neighborhood structure.* The neighborhoods here are sets of the form $N_{\hat{x},\hat{h}} \equiv \{\hat{x} + i\hat{h} | i \in \{0, \ldots, 10d\}\}$. Then $\mathcal{N} \equiv \bigcup_{\hat{x},\hat{h} \in \mathcal{Z}_p^m} N_{\hat{x},\hat{h}}$.
   (c) *Complexity of the neighborhood test.* Once again it turns out that the complexity of this test is within a constant factor of the complexity of

the test in characterization 2, i.e., $O(d^2)$ additions and subtractions and no multiplications (see appendix).

All characterizations above turn out to be robust. The robustness of characterization 1 is straightforward and omitted here (see, for instance, [Sud92]). The robustness of 4 follows from the work of [BFL91] (see also [BFLS91], [FGLSS91], [AS92], [Lun92]). The robustness of characterizations 2, 3, 5, and 6 are presented in §§4 and 5.

A typical robust characterization theorem for degree d polynomials in m variables over a finite field F would go as follows:

> There exists a δ_0 (which may be a function of d, m, and $|F|$) such that for $\delta \leq \delta_0$, if the fraction of neighborhoods where P satisfies the local constraints is at least $1 - \delta$, then P is ϵ-close to some degree-d polynomial (where ϵ is some function of δ).

An important parameter in determining the efficiency of a tester is the relationship between δ_0 and m, d, and $|F|$. For instance, if $\delta_0 = \frac{1}{dm \log |F|}$, then this implies that we will have to test that the local property holds for at least $dm \log |F|$ randomly chosen neighborhoods before we can satisfy ourselves that P is close to some polynomial. Our main thrust will be to get a theorem that holds for as high a δ_0 as possible.[2]

In what follows, we show first that characterization 5 above is robust with $\delta_0 = \Theta(\frac{1}{d^2})$. This proof gives a simple and efficient tester for the family of multivariate polynomials that works with $O(d^3)$ probes into f. Robustness of the characterizations in 2 and 3 follow as special cases. This bound on δ_0 is in contrast to the robustness of 4 that has an inherent dependency on m.

Next we show the robustness of characterization 6. The efficiency of this test is analyzed modulo the efficiency of a certain test for bivariate polynomials and is shown to be within a constant factor of the bivariate test. We also show that the efficiency of the bivariate test is $O(d)$, giving a test for multivariate polynomials that works with $O(d^2)$ probes into f.

4. A robust characterization of polynomial functions.

In this section, we prove the robustness of characterization 5. We consider a function (program) P mapping m variables from $\mathcal{Z}_p$ to $\mathcal{Z}_p$ and prove the following theorem.

THEOREM 4.1. *For $\delta_0 = \frac{1}{2(d+2)^2}$, if $P : \mathcal{Z}_p^m \mapsto \mathcal{Z}_p$ satisfies*

$$\delta \equiv \Pr_{x,h \in_R \mathcal{Z}_p^m} \left[P(x) \neq \sum_{i=1}^{d+1} \alpha_i P(x + i \cdot h) \right] \leq \delta_0,$$

then there exists a degree-d polynomial $g : \mathcal{Z}_p^m \mapsto \mathcal{Z}_p$ that is 2δ-close to P.

This theorem makes very minimal requirements on the field size required for its validity. The theorem is valid whenever characterization 5 holds, and Friedl and Sudan [FS94] have shown that this holds for $p \geq d + 2$—the smallest conceivable field size for which the test could be defined. We do not know of other testers that work with such a minimal requirement on the field size.

We define $g(x)$ to be $\text{maj}_{h \in \mathcal{Z}_p^m}\{\sum_{i=1}^{d+1} \alpha_i P(x + ih)\}$, where maj of a set is the function that picks the element occurring most often (choosing arbitrarily in the case of ties). First we show that g is 2δ-close to P. Later in this section, we show that g is a low-degree polynomial.

[2] A secondary parameter of interest is the relationship between ϵ and δ. In all the proofs that follow, we will only show that $\epsilon = 2\delta$. Actually, once such a result is shown, it can be shown again that any $\epsilon > \delta$ works.

LEMMA 4.2. *g and P agree on more than a $1-2\delta$ fraction of the inputs from Z_p^m.*

Proof. Consider the set of elements x such that $\Pr_h[P(x) = \sum_{i=1}^{d+1} \alpha_i P(x + i * h)] < \frac{1}{2}$. If the fraction of such elements is more than 2δ, then it contradicts the condition that $\Pr_{x,h}[\sum_{i=0}^{d+1} \alpha_i P(x+i*h) = 0] = \delta$. For all remaining elements, $P(x) = g(x)$. □

In the following lemmas, we show that the function g satisfies the interpolation formula for all x and h and is therefore a degree-d polynomial. We do this by first showing that for all x, $g(x)$ is equal to the interpolation of P at x by most offsets t. We then use this to show that the interpolation formula is satisfied by g for all x, h.

LEMMA 4.3. *For all $x \in Z_p^m$, $\Pr_h\left[g(x) = \sum_{i=1}^{d+1} \alpha_i P(x + i * h)\right] \geq 1 - 2(d+1)\delta$.*

Proof. Observe that $h_1, h_2 \in_R Z_p^m$ implies that when i and j do not equal 0,

$$x + i * h_1 \in_R Z_p^m \text{ and } x + j * h_2 \in_R Z_p^m$$

$$\Rightarrow \Pr_{h_1,h_2}\left[P(x + i * h_1) = \sum_{j=1}^{d+1} \alpha_j P(x + i * h_1 + j * h_2)\right] \geq 1 - \delta$$

$$\Rightarrow \Pr_{h_1,h_2}\left[P(x + j * h_2) = \sum_{i=1}^{d+1} \alpha_i P(x + i * h_1 + j * h_2)\right] \geq 1 - \delta.$$

Combining the two, we get

$$\begin{aligned} \Pr_{h_1,h_2}\left[\sum_{i=1}^{d+1} \alpha_i P(x + i * h_1)\right. &= \sum_{i=1}^{d+1}\sum_{j=1}^{d+1} \alpha_i\alpha_j P(x + i * h_1 + j * h_2) \\ &= \left.\sum_{j=1}^{d+1} \alpha_j P(x + j * h_2)\right] \\ &\geq 1 - 2(d+1)\delta. \end{aligned}$$

The lemma now follows from the observation that the probability that the same object is drawn twice from a set in two independent trials lower bounds the probability of drawing the most likely object in one trial. Suppose the objects are ordered so that p_i is the probability of drawing object i, and $p_1 \geq p_2 \geq \cdots$. Then the probability of drawing the same object twice is $\sum_i p_i^2 \leq \sum_i p_1 p_i = p_1$. □

LEMMA 4.4. *For all $x, h \in Z_p^m$, if $\delta \leq \frac{1}{2(d+2)^2}$, then $\sum_{i=0}^{d+1} \alpha_i g(x + i * h) = 0$ (and thus g is a degree-d polynomial* [dW70]).

Proof. Observe that, since $h_1 + ih_2 \in_R Z_p^m$, we have for all $0 \leq i \leq d+1$,

$$\Pr_{h_1,h_2}\left[g(x + i * h) = \sum_{j=1}^{d+1} \alpha_j P((x + i * h) + j * (h_1 + ih_2))\right] \geq 1 - 2(d+1)\delta.$$

Furthermore, we have for all $1 \leq j \leq d+1$

$$\Pr_{h_1,h_2}\left[\sum_{i=0}^{d+1} \alpha_i P((x + j * h_1) + i * (h + j * h_2)) = 0\right] \geq 1 - \delta.$$

Putting these two together, we get

$$\Pr_{h_1,h_2}\left[\sum_{i=0}^{d+1} \alpha_i g(x + i * h) = \sum_{j=1}^{d+1} \alpha_j \sum_{i=0}^{d+1} \alpha_i P((x + j * h_1) + i * (h + j * h_2)) = 0\right] > 0.$$

The lemma follows since the statement "$\sum_{i=0}^{d+1} \alpha_i g(x+i*h) = 0$" is independent of h_1 and h_2, and therefore if its probability is positive, it must be 1. □

Proof of Theorem 4.1. Theorem 4.1 follows from Lemmas 4.2 and 4.4. □

5. Efficient tester for polynomials. In this section, we prove the robustness of characterization 6. Recall that this characterization uses the collection of neighborhoods $\mathcal{N} = \{N_{x,h} | x, h \in \mathcal{Z}_p^m\}$, where $N_{x,h} = (x, x+h, \ldots, x+10dh)$. The following theorem shows that the efficiency of this characterization is $O(d)$, i.e., if a function satisfies the consistency test on all but a $O(\frac{1}{d})$ fraction of the neighborhoods, then it is close to a polynomial.

THEOREM 5.1. *There exists a constant c such that for* $0 \leq \delta \leq \frac{1}{cd}$, *if f is a function from* $\mathcal{Z}_p^m$ *to* $\mathcal{Z}_p$ *that satisfies the neighborhood consistency test on all but a* δ *fraction of the collection of neighborhoods* $\mathcal{N} = \{N_{x,h} | x, h \in \mathcal{Z}_p^m\}$ *(where* $N_{x,h} = \{x, x+h, \ldots, x+10dh\}$*), then there exists a polynomial* $g : \mathcal{Z}_p^m \to \mathcal{Z}_p$ *of total degree at most d such that* $d(f, g) \leq (1+o(1))\delta$ *(provided* $p > 10d$.*)*

In the rest of this section, we prove this theorem for the case $d \geq 1$. (The case $d = 0$ amounts to proving that f is a constant and is omitted as a straightforward exercise.)

Fix a function f that satisfies the neighborhood constraints on all but a δ fraction of the neighborhoods.

The proof follows the same basic outline as the one in §4, but in order to achieve the better efficiency, we use ideas that can be thought of in terms of error correction. Thus many of the steps that were quite simple in §4 require more work here. In §4, the function g was defined to be the value that occurs most often (for most h) when one looks at the evaluation at x of the unique polynomial that agrees with the values of f at $x+h, \ldots, x+(d+1)h$. Here we view the values of a polynomial at $x+h, \ldots, x+10dh$ as a codeword. Intuitively, the values of f at $x+h, \ldots, x+10dh$ will often have enough good information in it to allow us to get back to a correct codeword. The function g defined below can be thought of as the value that occurs most often (for most h) when one looks at the polynomial defined by the *error correction* of the values of f at $x, x+h, \ldots, x+10dh$ evaluated at x. We then show that g has the following properties:

1. $g(x) = f(x)$ with probability at least $1 - \delta - o(\delta)$ if x is picked randomly from $\mathcal{Z}_p^m$.
2. On every neighborhood $N_{x,h}$, g is described by a univariate polynomial of degree d.

Notice that characterization 6 now implies that g is a degree d polynomial.

Notation. For $x, h \in \mathcal{Z}_p^m$, we let $P_{x,h}(i)$ be (the unique) polynomial in i that satisfies $P_{x,h}(i) = f(x+ih)$ for at least $6d$ values of $i \in \{0, \ldots, 10d\}$. If no such polynomial exists, then $P_{x,h}$ is defined to be `error`.

$$\text{Let } g : \mathcal{Z}_p^m \mapsto \mathcal{Z}_p \text{ be } g(x) \equiv \text{plurality}_h\{P_{x,h}(0)\}$$

where the plurality is taken over P's that are not `error`.

In §4, it is shown that if one computes the value of a polynomial function at x by interpolating from the values of the function along offset h_1 that are in turn computed by interpolating from the values of the function along offset h_2, then one would get the same answer as if one had computed the value of the function at x by interpolating from the values of the function along offset h_2 that are in turn computed by interpolating from the values of the function along offset h_1. This is not hard to see because it turns out that an interpolation is a weighted summation and obtaining the identity amounts to changing the order of a double summation (see, for instance, Lemma 4.3). Here g is actually an interpolation of the *error correction* of the values of the function, which is no longer a simple algebraic function of the observed values. We repair the situation by constructing a bivariate polynomial $Q(i,j)$ that agrees with $f(x+i\cdot h_1+j\cdot h_2)$ for most values of i and j. This allows us to get back to simple interpolation where we work with the function $Q(i,j)$ rather than f. Lemma 5.2 shows when such a bivariate polynomial can be set up to agree with a matrix of values m_{ij}. Lemma 5.3 shows how to use this polynomial to simulate the effect of the interchange in the order of the summation.

The following lemma follows directly from the axis parallel characterization of polynomials.

LEMMA 5.2. *For $X, Y \subset \mathcal{Z}_p$ with $|X|, |Y| > d+2$, if $\{r_i\}_{i\in X}$ and $\{c_j\}_{j\in Y}$ are univariate (degree-d) polynomials such that for all $i \in X$ and $j \in Y$, $r_i(j) = c_j(i)$, then there exists a polynomial $Q(.,.)$ such that for all i,j $Q(i,j) = r_i(j) = c_j(i)$.*

LEMMA 5.3 (matrix polynomial lemma). *Given families of univariate degree-d polynomials $\{r_i\}_{i=0}^{10d}$ and $\{c_j\}_{j=0}^{10d}$ and a matrix $\{m_{ij}\}_{i=0,j=0}^{10d,10d}$ that satisfy the following:*

- *For 90% of the i's in $\{0,\dots,10d\}$, $r_i(j) = m_{ij}$ for all $j \in \{0,\dots,10d\}$.*
- *For 90% of the j's in $\{0,\dots,10d\}$, $c_j(i) = m_{ij}$ for all $i \in \{0,\dots,10d\}$.*

Then there exists a bivariate polynomial $Q(\cdot,\cdot)$ of degree d in each variable such that $\forall i_0, j_0 \in \{0,\dots,10d\}$ the following holds:

- *For at least 90% of the i's in $\{0,\dots,10d\}$, $Q(i,j_0) = m_{ij_0}$.*
- *For at least 90% of the j's in $\{0,\dots,10d\}$, $Q(i_0,j) = m_{i_0j}$.*

Proof. Let X be the set of good rows of M, i.e., those with the property that $r_i(j)$ equals $c_j(i)$ for all values of $j \in \{0,\dots,10d\}$. Similarly, let Y be the set of good columns. It can now be seen that the conditions of Lemma 5.2 are applicable, implying that there exists a polynomial $Q(i,j)$ such that $Q(i,j) = r_i(j) = c_j(i)$ for all $(i,j) \in X \times Y$, where $|X|$ and $|Y|$ are both at least $9d$. But for any $i \in X$, there exists a unique polynomial describing all the elements in its row and Q agrees with it on 90% of its elements. Thus, for $i \in X$, $Q(i,j) = m_{ij}$ for all $j \in \{0,\dots,10d\}$. In particular this holds for $j = j_0$, i.e., for all $i \in X$, $Q(i,j_0) = m_{ij_0}$. Similarly, by using the columns indexed by Y, one can show that $Q(i_0,j) = m_{i_0j}$ for all $j \in Y$. The lemma follows, since the cardinalities of X and Y are at least $9d$. □

The following lemmas are analogous to Lemmas 4.2–4.4 of §4. Lemma 5.4 and Corollary 5.5 roughly correspond to Lemma 4.3. Lemma 5.4 essentially states that the plurality in the definition of g is actually an overwhelming majority. This may be obtained by setting $i_0 = 0$ in the statement of the lemma. The slightly stronger statement used here is needed later. Lemma 5.6 is similar to Lemma 4.2 and shows that g and f agree at all but a $\delta + o(\delta)$ fraction of the places. Lemma 5.7 shows that g is a multivariate polynomial of degree d.

LEMMA 5.4. *There exists a constant c_1 such that for $\delta_1 = c_1\delta$, the following holds:*

$$\forall x \in F^m, i_0 \in \{0,\dots,10d\}, \quad \Pr_{h_1,h_2}[P_{x,h_1}(i_0) = P_{x+i_0h_1,h_2}(0)] \geq 1-\delta_1.$$

Proof. Pick $h_1, h_2 \in_R \mathcal{Z}_p^m$ and define $M = \{m_{ij}\}$ to be the matrix given by $m_{ij} = f(x+ih_1+jh_2)$. We show that M satisfies the conditions required by Lemma 5.3 (with $j_0 = 0$), with probability at least $1-\delta_1$. This suffices to prove the lemma, since this implies that the polynomial P_{x,h_1} is the polynomial $Q(i,j)$ restricted to $j = 0$ and that $P_{x+i_0h_1,h_2}$ is $Q(i_0,j)$. Thus $P_{x,h_1}(i_0) = P_{x+i_0h_1,h_2}(0) = Q(i_0,0)$.

Any row of the matrix, other than the 0th row, represents a random neighborhood (independent of x) and satisfies the neighborhood constraint with probability $1-\delta$. Thus with probability at least $1-10\delta$, we have that the fraction of rows that do not have a degree-d polynomial describing them is at most 0.1. An analogous argument can be made for the columns. Thus M satisfies the conditions required by Lemma 5.3 with probability at least $1 - 20\delta$. The lemma is satisfied with the choice of $c_1 = 20$. □

COROLLARY 5.5. *For $x \in \mathcal{Z}_p^m, i \in \{0, \ldots, 10d\}$, $\Pr_h[g(x+ih) = P_{x,h}(i)] \geq 1 - 2\delta_1$.*

Proof. Let B be the set of h's that violate $P_{x,h}(i) = \text{majority}_{h_1}\{P_{x+ih,h_1}(0)\}$. For all $h \notin B$, notice that $g(x+ih) = P_{x,h}(i)$. Also for h in B, the probability, for a randomly chosen h_1, that $P_{x+ih,h_1}(0) \neq P_{x,h}(i)$ is at least $\frac{1}{2}$. Thus with probability at least $\frac{|B|}{2p^m}$, we find that a randomly chosen pair (h, h_1) violates the condition $P_{x+ih,h_1}(0) = P_{x,h}(i)$. Applying Lemma 5.4, we get that $\frac{|B|}{p^m}$ is at most $2\delta_1$. □

We next show that g and f agree in most places.

LEMMA 5.6. *$d(f,g) \leq \delta(1+o(1))$.*

Proof. Let B' be the set of x's that satisfy $f(x) \neq P_{x,h}(0)$ for at least $1 - 2\delta_1$ fraction of the h's in $\mathcal{Z}_p^m$. Observe that due to Corollary 5.5, for all $x \notin B'$, $f(x)$ is the same as $g(x)$ ($g(x)$ can disagree with $P_{x,h}(0)$ on at most a $2\delta_1$ fraction of the h's). The size of B' as a fraction of $\mathcal{Z}_p^m$ can be at most $\frac{\delta}{1-2\delta_1}$. Thus we find that $d(f,g) \leq \frac{\delta}{1-2\delta_1} = \delta(1+o(1))$. □

Notation. For $x, h \in \mathcal{Z}_p^m$, we define $P_{x,h}^{(g)}(i)$ to be (the unique) polynomial in i that satisfies $P_{x,h}^{(g)}(i) = g(x+ih)$ for at least $9d$ values of $i \in \{0, \ldots, 10d\}$. If no such polynomial exists, then $P_{x,h}^{(g)}$ is defined to be `error`.

LEMMA 5.7. *There exists a constant γ such that if $\delta \leq \frac{1}{\gamma d}$ then $\forall x, h \quad g(x) = P_{x,h}^{(g)}(0)$.*

Proof. As in the proof of Lemma 5.4, we will pick a convenient matrix on which we will apply Lemma 5.3. This time the matrix of choice is obtained by picking $h_1, h_2 \in_R \mathcal{Z}_p^m$ and letting $m_{ij} = g(x + ih + j(h_1 + ih_2))$.

We will now show that Lemma 5.3 can be applied to this matrix with high probability (for $i_0 = j_0 = 0$). Observe that every row $\{m_{ij}\}_{j=0}^{10d}$ represents a random neighborhood containing the fixed point $x + ih$, and hence Corollary 5.5 implies that $P_{x+ih,h_1+ih_2}(j)$ agrees with m_{ij} for any choice of j with probability $1 - 2\delta_1$. Thus, for every i, with probability at least $1 - 2cd\delta_1$, $P_{x+ih,h_1+ih_2}(j)$ agrees with m_{ij} for all but $\frac{1}{cd}$ fraction of the j's. Thus with probability at least $1 - 22cd\delta_1$, this holds for at least 90% of the rows, including the row $i = 0$. By picking $c > 10$, we satisfy the conditions required of the rows in Lemma 5.3. A similar argument based on the columns shows that the conditions required of the columns are also true with probability $1 - 20cd\delta_1 - o(1)$ (all columns except for the 0th one represent random neighborhoods). Thus the conditions required for Lemma 5.3 are satisfied with probability at least $1 - 42cd\delta_1 - o(1)$.

Applying Lemma 5.3, we find that there exists a bivariate polynomial $Q(i,j)$ such that it agrees with m_{i0} for 90% of the i's. Thus $P_{x,h}^{(g)}(i) = Q(i,0)$. We now argue that $m_{00} = Q(0,0)$, and this will complete the proof, since $m_{00} = g(x)$.

By Lemma 5.3, we find that $m_{0j} = Q(0,j)$ for 90% of the j's, implying $P_{x,h_1}^{(g)}(j) = Q(0,j)$. By Corollary 5.5, we also find that $m_{00} = P_{x,h_1}(0)$ with probability at least $1 - 2\delta_1$. In order to show that this equals $Q(0,0)$, it now suffices to show that $P_{x,h_1}^{(g)}(\cdot) = P_{x,h_1}(\cdot)$.

This last part follows from the following observation: For $j \neq 0$, $x + jh_1$ is distributed uniformly over F^m, and thus with probability $1 - (1+o(1))\delta$, we have $g(x+jh_1) = f(x+jh_1)$ (by Corollary 5.5). Hence with probability at least $1 - 10\delta - o(1)$, $g(x+jh_1) = f(x+jh_1)$ for 90% of the j's. But both the polynomials $P_{x,h_1}(j)$

and $P^{(g)}_{x,h_1}(j)$ agree with $f(x+jh_1)$ and $g(x+jh_1)$ for 90% of the j's, respectively. Thus $P^{(g)}_{x,h_1}(\cdot)$ must agree with $P_{x,h_1}(\cdot)$ on at least 80% of the inputs, implying $P^{(g)}_{x,h_1}(\cdot) = P_{x,h_1}(\cdot)$.

Thus with probability at least $1-(42cd\delta_1+2\delta_1+10\delta+o(1))$ (over random choices of h_1 and h_2), the identity $g(x) = P^{(g)}_{x,h}(0)$ holds. But this event is deterministic (independent of h_1 and h_2) and hence if its probability is positive then it must always hold. If $\delta < 1/((20)(541)d)$, then $\delta_1 < 1/(541d)$ and thus the above probability is positive. □

Proof (*of Theorem* 5.1). Lemma 5.7 implies that along each line $l_{x,h}$, g can be described by a univariate polynomial of degree at most d. Characterization 6 can now be applied to infer that g is a polynomial of total degree at most d. From Lemma 5.6, we now know that f and g differ in at most $\delta(1+o(1))$ fraction of the places. This completes the proof. □

6. Self-testing polynomials. In this section, we complement the results of [BF90] and [Lip91] by showing how to construct a self-tester for any polynomial function. The results can also be generalized to give self-testers and self-correctors for functions in finite dimensional function spaces that are closed under shifting and scaling.

Previously, program testing was thought of as the following: *Pick a random input x and verify that $P(x) = f(x)$ by computing f via another program.* This method has two problems: first, it relies on believing the other program to be correct, and second, since testing is often done at runtime [BLR90], it negates the benefits of designing faster programs, since the computation time will be dominated by the computation time of the old program.

As in [BLR90], our testers are of a nontraditional form and use the robust characterization of the function being tested. The tester is given a short specification of the function in the form of properties that the function must have and verifies that these properties "usually" hold. We show that these properties are such that if the program "usually" satisfies these properties, then it is essentially computing the correct function.

6.1. Test sets. Given that a function computes a polynomial, we want a way of specifying that it is the correct polynomial. We do this by specifying the function value of the polynomial at a number of inputs. It is easy to see that the number of inputs required is exactly the number of inputs necessary to determine whether two degree-d polynomials are distinct. Since any two degree-d univariate polynomial functions can only agree on d points, it suffices to check whether or not the polynomial functions agree at *any* $d+1$ points to determine whether or not they are distinct. On the other hand, distinct multivariate polynomials can agree at an unbounded number of points. However, it is well known that there *exists* a set of $(d+1)^m$ points such that no two degree-d, m-variate polynomials can agree at all points in the set. We make the following definition.

DEFINITION 6.1. *We say that $\mathcal{T} = \{(x_1, y_1), \ldots, (x_t, y_t)\}$ is a (d,m)-*polynomial test set *if there is only one degree-d, m-variable polynomial f such that for all $i \in [1, \ldots, t]$, $f(x_i) = y_i$. A (d,m)-test set need only be of size $(d+1)^m$.*

When the number of variables is small, the provision that the value of the function is known on at least $(d+1)^m$ points is not very restrictive, since the degree is assumed to be small with respect to the size of the field. Suppose one has a program for the RSA function $x^3 \bmod q$. Traditional testing requires that the tester know the value of $f(x)$ for random values of x. Here one only needs to know the following simple and easy to generate specification: f is a degree-3 polynomial in one variable, and

$f(0) = 0, f(1) = 1, f(-1) = -1, f(2) = 8$. These function values are the same over any ring Z_q of size at least 9.

6.2. Testing algorithm. Our self-tester for a polynomial of degree d with m variables assumes that the specification of the polynomial is given by the value of the function on a (d, m)-polynomial test set.

THEOREM 6.2. *If f is a degree-d polynomial in m variables over $\mathcal{Z}_p$, and the value of f is given on a (d, m)-polynomial test set, then for $\epsilon \leq O(1/d^2)$, f has an $(\frac{\epsilon}{2(d+2)}, 4\epsilon)$-self-tester on $\mathcal{Z}_p$ with $O((d+1)^m/\epsilon + d \cdot \max(d^2, \frac{1}{\epsilon}))$ calls to P.*

The self-testing is done in two phases, one verifying that the program is essentially computing some degree-d polynomial function g and the other verifying that the g is the correct polynomial function by verifying that g (rather than P) is correct on the polynomial test set.

We now give the self-testing program that is used to prove Theorem 6.2.

For simplicity, in the description of our self-testing program, we assume that whenever the self-tester makes a call to P, it verifies that the answer returned by P is in the proper range, and if the answer is not in the proper range, then the program notes that there is an error.

We use $x \in_R Z_p^m$ to denote that x is chosen uniformly at random in Z_p^m.

```
PROGRAM POLYNOMIAL-SELF-TEST ($P, \epsilon, \beta, \mathcal{T} = ((x_1, f(x_1)), \ldots, (x_t, f(x_t)))$)

Degree Test
    Repeat $\Theta(\frac{1}{\epsilon} \log(1/\beta))$ times
        Pick $x, h \in_R Z_p^m$ and test that $\sum_{i=0}^{d+1} \alpha_i P(x + i * h) = 0$
    Reject $P$ if the test fails more than an $\epsilon$ fraction of the time.

Equality Test
    For $j$ going from 1 to $t$ do
        Repeat $\Theta(\log(d/\beta))$ times
            Pick $h \in_R Z_p^m$ and test that $f(x_j) = \sum_{i=1}^{d+1} \alpha_i P(x_j + i * h)$.
        Reject $P$ if the test fails more than 1/4th of the time.
```

6.3. Correctness of algorithm: Notation. Let $\delta \equiv \Pr_{x,h}[\sum_{i=0}^{d+1} \alpha_i P(x + i * h) \neq 0]$. We say program P is ϵ-good if $\delta \leq \frac{\epsilon}{2}$ and $\forall j \in \{1, \ldots, t\}$, $\Pr_h[f(x_j) = \sum_{i=1}^{d+1} \alpha_i P(x_j + i * h)] \geq \frac{3}{4}$. We say P is ϵ-bad if either $\delta > 2\epsilon$ or if $\exists j$ such that $\Pr_h[f(x_j) = \sum_{i=1}^{d+1} \alpha_i P(x_j + i * h)] < \frac{1}{2}$. (Note that there are programs that are neither ϵ-good or ϵ-bad.)

The following lemma is easy to prove.

LEMMA 6.3. *With probability at least $1 - \beta$, an ϵ-good program is passed by Polynomial-Self-Test. With probability at least $1 - \beta$, an ϵ-bad program is rejected by Polynomial-Self-Test.*

It is easy to see that if a program P $\frac{\epsilon}{2(d+2)}$-computes f, then it is ϵ-good. On the other hand, we need to show that if P does not 4ϵ-compute f, then it is ϵ-bad. We show the contrapositive, i.e., that if P is not ϵ-bad, then it 4ϵ-computes f.

If P is not ϵ-bad, then $\delta \leq 2\epsilon$. Under this assumption, we show that there exists a function g with the following properties:

1. $g(x) = P(x)$ for most x.
2. $\forall x, t, \sum_{i=0}^{d+1} \alpha_i g(x + it) = 0$, and thus g is a degree-d polynomial.
3. $g(x_j) = f(x_j)$ for $j \in \{0, 1, \ldots, d\}$.

The function g is as defined in the previous section on robust characterizations, and properties 1 and 2 follow from the lemmas proved there. In order to show property 3, we also have the following lemma.

LEMMA 6.4. *$g(x_j) = f(x_j)$.*

Proof. The proof follows from the definition of g and the fact that P is not ϵ-bad. □

THEOREM 6.5. *The program Polynomial-Self-Test is an $(\frac{\epsilon}{2(d+2)}, 4\epsilon)$-self-testing program for any degree-d polynomial function over Z_p^m specified by its values at any (d,m)-polynomial test set $\mathcal{T}$, if $\epsilon \leq \frac{1}{4(d+2)^2}$.*

Proof. The proof follows from Lemmas 6.3, 4.4, and 6.4. □

7. Locally testable codes.

In this section, we introduce some definitions related to coding and show the implications of low-degree testing to generating codes with nice properties.[3] We start by describing some standard parameters associated with error-correcting codes.

An n-letter string over the alphabet Σ is an element of Σ^n. Given a string $w \in \Sigma^n$, the ith character of w is denoted w_i. Given strings $w, w' \in \Sigma$, the relative distance between w and w', denoted $d(w, w')$ is the fraction of indices $i \in \{1, \ldots, n\}$ where $w_i \neq w'_i$. (Here onwards we will drop the term relative from the description of this parameter.)

DEFINITION 7.1 (error-correcting code). *A (k, n, Δ, a)-code consists of an alphabet Σ such that $\log|\Sigma| = a$ and a function $C : \Sigma^k \to \Sigma^n$, such that for any two strings $m, m' \in \Sigma^k$, the distance between $C(m)$ and $C(m')$ is at least Δ.*

For the purposes of this section, we will restrict our attention to error-correcting codes within a small range of the above parameters which are interesting for the applications to probabilistically checkable proofs. We call these the *good* codes. Such codes need to have constant relative distance. The encoded message is allowed to be much larger than the original message size, as long as the final length is polynomially bounded. Perhaps the most interesting aspect is the alphabet size. While the ultimate goal would be to get codes which work over a constant-sized alphabet, getting an alphabet size which is significantly smaller than the message size (smaller than any nonconstant polynomial) turns out to be an important intermediate goal. Here we choose this parameter to be polylogarithmic in the message size.

DEFINITION 7.2 (good code). *A family of codes $\{C_i\}$ with parameters $(k_i, n_i, \Delta_i, a_i)$ is* good *if $k_i \to \infty$, n_i is upper bounded by some polynomial in k_i, $\Delta_i > 0$, and a_i is upper bounded by some function growing as* polylog(k_i).

A wide variety of codes described in practice satisfy the properties required of a good code. In particular we describe the polynomial codes.

DEFINITION 7.3 (polynomial codes). *Fix some $\epsilon > 0$. The* polynomial codes *$\{P_m\}$ are chosen by letting $d = \lceil m^{1+\epsilon} \rceil$ and picking a finite field F of size between $10d$ and $20d$. The code achieves $k_m = \binom{m+d}{m}$ and $n_m = |F|^m$ over the alphabet F and works as follows: The message is viewed as specifying the coefficients of a degree-d polynomial in m variables and the encoding consists of the value of this polynomial at all inputs.*

It may be verified that $\{P_m\}$ forms a good code with distance at least 0.9. In what follows, we will describe how this family of codes and a related code have extremely "good" local checkability properties. The following definition formalizes the notion of local checkability. Informally, the definition expects that by probing a string in just

[3] These definitions are motivated by subsequent work in the area of proof checking where our tester has found applications, most notably that of [ALMSS92].

p (randomly chosen) letters, the verifier can test if it close to a valid codeword and if not rejects it with probability at least δ.

DEFINITION 7.4 (locally testable code). *For a positive integer p and a positive real number δ, an (n,k,Δ,a)-code C over the alphabet Σ is* (p,δ)-locally testable *if the following exist:*

- *a probability space Ω which can be* efficiently *sampled,*
- *functions $q_1, q_2, \dots, q_p : \Omega \to \{1,\dots,n\}$, and*
- *a boolean function $V : \Omega \times \Sigma^p \to \{0,1\}$,*

with the property that for all $w \in \Sigma^n$, if

$$\Pr_{r\in\Omega}\left[V(r, w_{q_1(r)}, \dots, w_{q_p(r)}) = 0\right] < \delta$$

then there exists a (unique) string $m \in \Sigma^k$ such that $d(w, C(m)) < \Delta/2$. Conversely, if $w = C(m)$ for some m, then $V(r, w_{q_1(r)}, \dots, w_{q_p(r)}) = 1$ for all $r \in \Omega$.

Before we describe the kind of locally checkable codes that our testers provide, we attempt to motivate the definition above by showing that (seemingly minor) modifications of the above definitions yield important concepts in proof checking—namely, probabilistically checkable proofs. We consider especially probabilistically checkable proofs over a large alphabet in which the number of alphabets that a verifier is allowed to probe is a parameter. This concept is an important ingredient in the recursive construction of probabilistically checkable proofs [AS92], [ALMSS92], [BGLR93] and is also of independent interest in complexity theory [LS91], [FL92a]. The original definition of probabilistically checkable proofs is due to [AS92] based on an implicit notion in [FGLSS91]. A very closely related notion—that of holographic proofs—appears in the work of [BFLS91]. The particular choice of parameters made in the following definition is due to [BGLR93].

DEFINITION 7.5 (PCP). *Given functions $r, p, a, \delta : \mathcal{Z}^+ \to \mathcal{Z}^+$, a language $L \subset \{0,1\}^*$ is said to be in PCP$[r,p,a,\delta]$ if there exists a polynomially growing function $n(l)$ and an alphabet Σ of size $a(l)$ such that for all integers $l > 0$ the following exist:*

- *a probability space Ω which can be sampled using $r(l)$ bits,*
- *functions $q_1, q_2, \dots, q_{p(k)} : \Omega \to \{1,\dots,n(l)\}$,*
- *a boolean function $V : \{0,1\}^l \times \Omega \times \Sigma^p \to \{0,1\}$,*

with the property that for all $x \in \{0,1\}^l$, if $w \in \Sigma^{n(l)}$ satisfies

$$\Pr_{r\in\Omega}\left[V(x, r, w_{q_1(r)}, \dots, w_{q_p(r)}) = 0\right] < \delta,$$

then $x \in L$. Conversely, if $x \in L$, then there exists $w \in \Sigma^n(l)$ such that for all $r \in \Omega$, $V(x, r, w_{q_1(r)}, \dots, w_{q_p(r)}) = 1$.

It turns out that there is strong correlation between PCP$[\log, p, \text{polylog}, \delta]$ and good codes which are (p,δ)-locally checkable. In particular, the codes we describe next translate into such probabilistically checkable proofs.

The robust characterization of polynomials described in Theorem 5.1 shows that the polynomial codes are $(d+2, \Omega(1/d))$-locally testable. Observe further that for the polynomial codes, the growth of d is polylogarithmic in k. It seems that the approach above cannot hope to give codes which are testable using fewer than $\Omega(d)$ probes. However, this is not the case. We describe next a simple way of modifying the codes so as to give codes with appreciably better local testability. These codes are obtained by observing that the codes we have constructed so far use a much smaller alphabet size than necessary for "goodness."

DEFINITION 7.6 (polynomial-line codes). *Fix some $\epsilon > 0$. The* polynomial-line codes *$\{L_m\}$ are chosen by letting $t = \lceil m^{1+\epsilon} \rceil$ and picking a finite field F of size* *between 10d and 20d. The code achieves* $k_m = \binom{m+d}{m}/(d+1)$ *and* $n_m = |F|^{2m}$ *over the alphabet* F^{d+1}. *As in the polynomial codes, the message again consists of* $\binom{m+d}{d}$ *field elements and is viewed as a degree-d polynomial specified by its coefficients. Given a message polynomial p, the codeword is constructed as follows: For every pair of field elements* $\hat{x}, \hat{h} \in F^m$, *let* $l_{\hat{x},\hat{h}}$ *be the line through* $\hat{x}$ *with offset* $\hat{h}$ *as in characterization 3. p restricted to* $l_{\hat{x},\hat{h}}$ *is a univariate polynomial of degree d. Let* $C_{\hat{x},\hat{h}} \in F^{d+1}$ *be the vector of coefficients of this univariate polynomial. The codeword consists of* $\{C_{\hat{x},\hat{h}}\}_{\hat{x},\hat{h}\in F^m}$.

It is easy to see that the polynomial-line codes are also good codes. The proof of Theorem 5.1 can be transformed to show that the polynomial-line codes are locally testable with a constant number of probes. More specifically, the following can be shown.

PROPOSITION 7.7. *The polynomial-line codes are* $(2, \Omega(1/d))$*-locally testable.*

Better analysis of some portions of our proof yields even better statements about the polynomial-line codes. This is described in the next section.

8. Conclusions.

There has been a spate of results about low-degree tests in the last few years. A brief listing includes the low-degree test of [BFL91] and [Lun92], which was the first test for multivariate polynomials, the results of [BFLS91] and [FGLSS91], obtained independently from and concurrently with ours (from [GLRSW91] and [RS92]), and subsequent works [AS92], [ALMSS92], [FHS94], [PS94], and [FS94]. Here we summarize some of their achievements along with a comparison with our results. We start by distinguishing the merits of our tester from those of [BFL91] and [BFLS91].

8.1. Program checking.

The test of [BFL91] and [Lun92] in the program-checking setting allows the self-tester to be convinced that the program is computing a multivariate polynomial function of low degree in polynomial time. However, the tests are somewhat complicated to perform, because they involve the reconstruction of a univariate polynomial, given its values at a number of points (which in turn requires multiplications and matrix inversions), and later the evaluation of the reconstructed polynomial at random points. If the given function is a function of a single variable, then the tester of [BFL91] and [Lun92] is no simpler than a program evaluating the polynomial. Therefore it does not have the "little-oh" property defined in [BK89] nor is it *different* from the program evaluating the polynomial, in the sense defined by [BLR90], and it does not give a self-tester or checker. Our test in contrast is *different*, since it requires no multiplications to perform the test.

8.2. Relationship with proof checking.

The low-degree tester forms a crucial ingredient in the recent results on proof checking. Our result from §4 gives a very simple proof of one of the relatively hard parts of the proof of MIP=NEXPTIME shown by [BFL91]. The hardness of the analysis of the tester of [BFL91] (and its simplifications; see, for instance, [FGLSS91]) is in their need to rely on the isoperimetric properties of the m-dimensional grid. Our proof, on the other hand, does not seem to require any combinatorics and is instead based on elementary algebraic/probabilistic techniques. This difference may be explained as follows: The success of the test does indeed depend on the isoperimetric properties of a graph related to the neighborhood structure. In the case of the test of [BFL91], this graph turns out to be in the m-dimensional grid. In our case, the underlying graph turns out to be a complete graph. This graph is obviously much easier to analyze because its properties and hence the proof are devoid of any combinatorial statements.

We now describe some of the subsequent results and the role of our tester in these results. The contrast is described in terms of locally testable codes.

8.3. Locally testable codes. The low-degree test described in [BFL91] and [BFLS91] gives rise to good codes which also have nice local checkability property. A sequence of improvements [BFL91], [BFLS91], [FGLSS91] culminated in the work of [AS92], which achieves asymptotically optimal bounds for such codes by showing that they are $(2, \Omega(1/m))$-locally testable. The highlight of [AS92] is that the locality bounds are independent of the degree of the polynomial that they work with. However, the dependence of δ on m is inherent for such codes and $\delta \to 0$ as $m \to \infty$. The polynomial-line codes described in §7 seem to have no inherent reason why δ should go to zero. This turns out to be indeed the case. In [ALMSS92], it is observed that a combination of the analysis of [AS92] and that of §5 implies that there exists a constant $\delta > 0$ such that the polynomial-line codes are $(2, \delta)$-locally testable, provided that the field F is of cardinaltity at least d^2. As mentioned in §7, this translates into a proof of NP $\subset$ PCP[log, $O(1)$, polylog, $\Omega(1)$] in [ALMSS92]. By employing the technique of recursive proof checking, due to [AS92], on such proof systems [ALMSS92] go on to prove that NP $\subset$ PCP[log, $O(1)$, $O(1)$, $\Omega(1)$]. The local testability of the polynomial-line codes has been further improved in two ways recently. [PS94] have shown that the polynomial-line codes are $(2, \delta)$-locally checkable over linear-sized fields as well, for some $\delta > 0$. In a different direction, it is shown in [FS94] that the polynomial-line codes are $(2, \delta)$-locally checkable for all $\delta < \frac{1}{8}$.

Appendix.

A.1. Evenly spaced points. The following algorithm may be used to test if a function $f^{(0)}$ on m evenly spaced points—$x, x+h, \ldots, x+(m-1)h$—(where $m > d+1$) agrees with a degree-d polynomial.

```
for $i = 1$ to $d + 1$ do
    for $j = 1$ to $m - i$
        $f^{(i)}(x + jh) = f^{(i-1)}(x + (j + 1)h) - f^{(i-1)}(x + jh)$
    endfor
endfor
verify $f^{(d+1)}(x + jh) = 0$, for all $j \in \{0, \ldots, m - d - 2\}$.
```

The correctness of this algorithm follows from the following well-known fact:

FACT A.1. *$f^{(i)}(x)$ is a degree-$(d-i)$ polynomial if and only if $f^{(i-1)}$ is a degree-$(d-i+1)$ polynomial.*

(Follows from the fact that $f^{(i)}$ acts as the discrete derivative of $f^{(i-1)}$.)

This implies that $f^{(d)}$ is a constant if and only if $f^{(0)}$ is a degree-d polynomial, implying in turn that $f^{(d+1)}$ is identically zero if and only if $f^{(0)}$ is a degree-d polynomial. Observe further that the algorithm performs $O(md)$ additions and subtractions and no multiplications. Finally, it can also be checked that in case $m = d + 2$, then algorithm simply verifies that $\sum_{i=0}^{d+1} \alpha_i f^{(0)}(x + ih) = 0$, where $\alpha_i = (-1)^{i+1}\binom{d+1}{i}$.

A.2. Characterizations.

LEMMA A.2 (axis-parallel lines). *$f : \mathcal{Z}_p^m \mapsto \mathcal{Z}_p$ is a polynomial in m variables of degree at most d in each variable if and only if for all $i \in \{1, \ldots, m\}$, $\beta_j \in \mathcal{Z}_p$ $(j \neq i)$, $f(\beta_1, \ldots, \beta_{i-1}, x_i, \beta_{iH}, \ldots, \beta_m)$ is a polynomial in x_i of degree at most d.*

Proof [*Sketch*]. It is clear that every polynomial of degree d in each variable restricted to axis-parallel lines behaves as a univariate polynomial of degree d. The other direction can be proved by induction on m. The base case $m = 1$ is obvious. For general $m > 1$, let $f_i(x_1, \ldots, x_{m-1})$ be the function $f(x_1, \ldots, x_{m-1}, i)$. By induction f_i is a polynomial of degree d in $m - 1$ variables. Now consider the function $h(x_1, \ldots, x_m) \equiv \sum_{i=0}^{d} \delta_i^{(d)}(x_m) f_i(x_1, \ldots, x_{m-1})$ (where $\delta_i^{(d)}$ is the unique polynomial of degree d in one variable that is 1 at $x_m = i$ and 0 for other values of $x_m \in \{0, \ldots, d\}$).

It is clear by construction that h is a polynomial of degree at most d in each variable. We now argue that f and h are identical. Fix $x_1 = \beta_1, \ldots, x_{m-1} = \beta_{m-1}$. It is clear that $h(x_1, \ldots, x_m) = f(x_1, \ldots, x_m)$ for $x_m \in \{0, \ldots, d\}$. Moreover, both h and f are degree-d polynomials in x_m which agree at $d + 1$ places. Hence f and h must agree at all values of x_m. Since this held for any choice of β_i's, f and h agree everywhere. □

LEMMA A.3 (general lines). *For $p \geq 2d + 1$, $f : \mathcal{Z}_p^m \mapsto \mathcal{Z}_p$ is a polynomial in m variables of total degree at most d if and only if $\forall \hat{x}, \hat{h} \in \mathcal{Z}_p^m$, $f(\hat{x} + t \cdot \hat{h})$ is a univariate polynomial in t of degree at most d.*

Proof. It is clear that every polynomial restricted to lines must become a degree-d polynomial in the parameter describing the line. Here we prove the other direction of the characterization. We first observe that since the set of all lines includes the axis parallel lines, we can use Lemma A.2 to show that f is a polynomial in m variables with degree at most d in each variable. Having got this weak characterization, we will now strengthen this to a tighter one. By induction on the number of variables, we can assume that f restricted to any value of the last variable x_m is a polynomial of total degree at most d in the variables $x_1, \ldots, x_{m-1}$. Thus f becomes a function in x_1 through x_m of total degree $d' \leq 2d$.

Assume for contradiction that $d' > d$. Now consider the function $f(t \cdot \hat{h})$ for $\hat{h} \in_R \mathcal{Z}_p^m$. The coefficient of $t^{d'}$ is a polynomial in $\hat{h}$ of degree d' which with probability at least $1 - \frac{d'}{p}$ should be nonzero. (Note that to make this probability positive, we need $2d < p$.) Thus f restricted to this line is a polynomial of degree $d' > d$, which violates the given condition on f. □

Acknowledgments. We are very grateful to Avi Wigderson for suggesting that our tester in §4 can be made more efficient, as well as his technical help in proving the theorems of §5. We are also very grateful to Sasha Shen for pointing out that the tester given in [GLRSW91] works for multivariate polynomials. In particular, characterization 3 and its relevance to our test are due to him. We are grateful to Dick Lipton for illuminating conversations on the use of the testers presented here, and to Mike Luby, Shafi Goldwasser, and Umesh Vazirani for technical suggestions. We would also like to thank Dieter van Melkebeek and the anonymous referees for pointing out numerous errors in earlier versions.

REFERENCES

[AHK] L. ADLEMAN, M. HUANG, AND K. KOMPELLA, *Efficient checkers for number-theoretic computations*, Inform. and Comput., to appear.

[ALMSS92] S. ARORA, C. LUND, R. MOTWANI, M. SUDAN, AND M. SZEGEDY, *Proof verification and the intractability of approximation problems*, in Proc. 33rd IEEE Symposium on Foundations of Computer Science, IEEE Press, Piscataway, NJ, 1992, pp. 14–23.

[AS92] S. ARORA AND S. SAFRA, *Probabilistic checking of proofs: A new characterization of NP*, in Proc. 33rd Annual IEEE Symposium on Foundations of Computer Science, IEEE Press, Piscataway, NJ, 1992, pp. 2–13.

[Bab93] L. BABAI, *Transparent (holographic) proofs*, in Proc. 10th Annual Symposium on Theoretical Aspects of Computer Science, Lecture Notes in Comput. Sci., 665 (1993), pp. 525–533.

[BF90] D. BEAVER AND J. FEIGENBAUM, *Hiding instances in multioracle queries*, in Proc. 7th Annual Symposium on Theoretical Aspects of Computer Science, Lecture Notes in Comput. Sci., 415 (1990), pp. 37–48.

[BF91] L. BABAI AND L. FORTNOW, *Arithmetization: A new method in structural complexity theory*, Comput. Complexity, 1 (1991), pp. 41–66.

[BFL91] L. BABAI, L. FORTNOW, AND C. LUND, *Non-deterministic exponential time has two-prover interactive protocols*, Comput. Complexity, 1 (1991), pp. 3–40.

[BFLS91] L. BABAI, L. FORTNOW, L. LEVIN, AND M. SZEGEDY, *Checking computations in polylogarithmic time*, in Proc. 23rd Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1991, pp. 21–31.

[BGLR93] M. BELLARE, S. GOLDWASSER, C. LUND, AND A. RUSSELL, *Efficient probabilistically checkable proofs*, in Proc. 25th Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1993, pp. 294–304.

[BK89] M. BLUM AND S. KANNAN, *Program correctness checking ... and the design of programs that check their work*, in Proc. 21st Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1989, pp. 86–97.

[BLR90] M. BLUM, M. LUBY, AND R. RUBINFELD, *Self-testing/correcting with applications to numerical problems*, in Proc. 22nd Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1990, pp. 73–83; J. Comput. System Sci., 47 (1993), pp. 549–595.

[Blu88] M. BLUM, *Designing programs to check their work*, Tech. report TR-88-009, International Computer Science Institute, Berkeley, CA, 1988.

[dW70] B. VAN DER WAERDEN, *Algebra*, vol. 1, Frederick Ungar, New York, 1970.

[FGLSS91] U. FEIGE, S. GOLDWASSER, L. LOVASZ, S. SAFRA, AND M. SZEGEDY, *Approximating clique is almost NP-complete*, in Proc. 32nd IEEE Symposium on Foundations of Computer Science, IEEE Press, Piscataway, NJ, 1991, pp. 2–12.

[FHS94] K. FRIEDL, Z. HATSAGI, AND A. SHEN, *Low-degree tests*, in Proc. 5th Annual ACM–SIAM Symposium on Discrete Algorithms–Society for Industrial and Applied Mathematics, Philadelphia, 1994, pp. 57–64.

[FL92a] U. FEIGE AND L. LOVASZ, *Two-prover one-round proof systems: Their power and their problems*, in Proc. 24th ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1992, pp. 733–744.

[FS94] K. FRIEDL AND M. SUDAN, *Improvements to total degree tests*, in Proc. 3rd Israel Symposium on Theory of Computing and Systems, IEEE Press, Piscataway, NJ, 1995, pp. 190–198.

[GLRSW91] P. GEMMELL, R. LIPTON, R. RUBINFELD, M. SUDAN, AND A. WIGDERSON, *Self-testing/correcting for polynomials and for approximate functions*, in Proc. 23rd Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1991, pp. 32–42.

[Kan90] S. KANNAN, *Program result checking with applications*, Ph.D. thesis, University of California at Berkeley, Berkeley, CA, 1990.

[Lip91] R. LIPTON, *New directions in testing*, Distributed Computing and Cryptography, DIMACS Series in Discrete Math and Theoretical Computer Science, American Mathematical Society, Providence, RI, 1991, pp. 191–202.

[LS91] D. LAPIDOT AND A. SHAMIR, *Fully parallelized multi prover protocols for NEXP-TIME*, in Proc. 32nd IEEE Symposium on Foundations of Computer Science, IEEE Press, Piscataway, NJ, 1991, pp. 13–18.

[Lun92] C. LUND, *The Power of Interaction*, ACM Distinguished Dissertations series, MIT Press, Cambridge, MA, 1992.

[Nao92] M. NAOR, *personal communication*, April 1992.

[PS94] A. POLISHCHUK AND D. SPIELMAN, *Nearly-linear size holographic proofs*, in Proc. 26th Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1994, pp. 194–203.

[RS92] R. RUBINFELD AND M. SUDAN, *Testing polynomial functions efficiently and over rational domains*, in Proc. 3rd Annual ACM–SIAM Symposium on Discrete Algorithms, Society for Industrial and Applied Mathematics, Philadelphia, 1992, pp. 23–43.

[Rub90] R. RUBINFELD, *A mathematical theory of self-checking, self-testing and self-correcting programs*, Ph.D. thesis, University of California at Berkeley, Berkeley, CA, 1990.

[She91] A. SHEN, personal communication, May 1991.

[Sud92] M. SUDAN, *Efficient checking of polynomials and proofs and the hardness of approximation problems*, Ph.D. thesis, University of California at Berkeley, Berkeley, CA, 1992.

SIAM J. COMPUT.
Vol. 25, No. 2, pp. 272–289, April 1996

GENOME REARRANGEMENTS AND SORTING BY REVERSALS*

VINEET BAFNA† AND PAVEL A. PEVZNER‡

Abstract. Sequence comparison in molecular biology is in the beginning of a major paradigm shift—a shift from *gene* comparison based on local mutations (i.e., insertions, deletions, and substitutions of nucleotides) to *chromosome* comparison based on global rearrangements (i.e., inversions and transpositions of fragments). The classical methods of sequence comparison do not work for global rearrangements, and little is known in computer science about the edit distance between sequences if global rearrangements are allowed. In the simplest form, the problem of gene rearrangements corresponds to *sorting by reversals*, i.e., sorting of an array using reversals of arbitrary fragments. Recently, Kececioglu and Sankoff gave the first approximation algorithm for sorting by reversals with guaranteed error bound 2 and identified open problems related to chromosome rearrangements. One of these problems is Gollan's conjecture on the reversal diameter of the symmetric group. This paper proves the conjecture. Further, the problem of expected reversal distance between two random permutations is investigated. The reversal distance between two random permutations is shown to be very close to the reversal diameter, thereby indicating that reversal distance provides a good separation between related and nonrelated sequences in molecular evolution studies. The gene rearrangement problem forces us to consider reversals of *signed permutations*, as the genes in DNA could be positively or negatively oriented. An approximation algorithm for signed permutation is presented, which provides a performance guarantee of $\frac{3}{2}$. Finally, using the signed permutations approach, an approximation algorithm for sorting by reversals is described which achieves a performance guarantee of $\frac{7}{4}$.

Key words. computational molecular biology, sorting by reversals, genome rearrangements

AMS subject classifications. 68Q25, 68Q05

1. Introduction.

Genus *Lobelia* comprises over 350 species that range from small, slender herbs to woody, giant-rosette plants. Figure 1 presents the order of genes in *Tobacco* and *Lobelia fervens* chloroplast genomes with a hypothetical sequence of rearrangement events (Knox et al. [KDP93]) during evolution of *Lobelia fervens* from a tobacco-like ancestral genome.

It is not so easy to verify that the evolutionary events shown in Fig. 1 represent the *shortest* series of reversals transforming the *Tobacco* permutation into the *Lobelia fervens* permutation. In fact, Theorem 2 of this paper indicates that the shortest sequence of rearrangement events contains just 4 reversals, shown in Fig. 2 (however, for the case of *signed* permutations (see below) the evolutionary events presented in Fig. 1 do represent the shortest series of reversals).

With the advent of large-scale DNA mapping and sequencing, the number of *genome comparison* problems similar to the one presented in Fig. 1 is rapidly growing in different areas, including evolution of plant cpDNA (Raubeson and Jansen [RJ92], Hoot and Palmer [HP94]) and mtDNA (Palmer and Herbon [PH88], Fauron and Havlik [FH89]), animal mtDNA (Hoffman et al. [HBB92], Sankoff et al. [SLA92]), virology (Koonin and Dolya [K93], Hannenhalli et al. [HCKP95]), *Drosophila* genetics (Whiting et al. [WPFJ89]), and comparative physical mapping (Lyon [L88]). Genome comparison has certain merits and demerits as compared to classical *gene* comparison.

* Received by the editors June 17, 1993; accepted for publication (in revised form) August 22, 1994. A preliminary version of this paper appeared in Proc. 34th IEEE Symposium on the Foundations of Computer Science, 1993, pp. 148–157.

† Computer Science Department, The Pennsylvania State University, University Park, PA 16802.

‡ Computer Science Department, The Pennsylvania State University, University Park, PA 16802 (pevzner@cs.psu.edu).

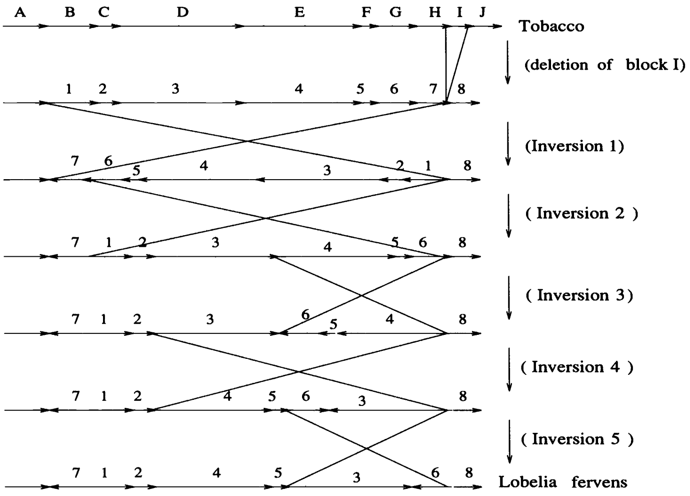


FIG. 1. *Evolution of* Lobelia fervens.

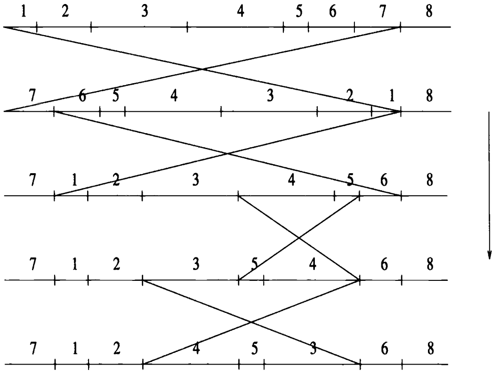


FIG. 2. *An optimal reversal sequence for* $\pi = 71245368$.

Genome comparison ignores actual DNA sequences of genes, while gene comparison ignores gene order. The ultimate goal would be to combine merits of both genome and gene comparison in a single algorithm. However, until recently, studies of genome rearrangements were based on heuristic methods, and there were no algorithms to analyze gene orders in molecular evolution. Kececioglu and Sankoff found an algorithm for reversal distance with guaranteed error bound 2 and raised a spectrum of open problems motivated by genome rearrangements [KS93]. The present paper solves some of them.

In the problem we consider, the order of genes in two organisms is represented by permutations $\pi = (\pi_1\pi_2\dots\pi_n)$ and $\sigma = (\sigma_1\sigma_2\dots\sigma_n)$. A *reversal* $\rho(i,j)$ of an interval$[i,j]$ is the permutation

$$\begin{pmatrix} 1 & 2 & \dots & i-1 & \mathbf{i} & \mathbf{i+1} & \dots & \mathbf{j-1} & \mathbf{j} & j+1 & \dots & n \\ 1 & 2 & \dots & i-1 & \mathbf{j} & \mathbf{j-1} & \dots & \mathbf{i+1} & \mathbf{i} & j+1 & \dots & n \end{pmatrix}.$$

Clearly $\pi \cdot \rho$ has the effect of reversing genes $\pi_i, \pi_{i+1}, \dots, \pi_j$.

Given permutations π and σ, the *reversal distance problem* is to find a series of reversals $\rho_1, \rho_2, \ldots, \rho_t$ such that $\pi \cdot \rho_1 \cdot \rho_2 \cdots \rho_t = \sigma$ and t is minimum. We call t the *reversal distance* between π and σ. Note that reversal distance between π and σ equals the reversal distance between $\sigma^{-1}\pi$ and the *identity* permutation $\imath$. *Sorting* π *by reversals* is the problem of finding reversal distance $d(\pi)$, between π and $\imath$.

Reversals generate the *symmetric group* S_n. Given an arbitrary permutation π from S_n, we seek a shortest product of *generators* $\rho_1 \cdot \rho_2 \cdots \rho_t$ that equals π. Even and Goldreich [EG81] show that given a set of generators of a permutation group and a permutation π, determining the shortest product of generators that equals π is NP-hard. Jerrum [J85] proves that the problem is PSPACE-complete, and remains so, when restricted to two generators. In our problem, the generator set is fixed. However, Kececioglu and Sankoff [KS93] conjecture that sorting by reversals is NP-complete.

Gates and Papadimitriou [GP79] studied a similar *sorting by prefix reversals* problem (also known as *pancake-flipping problem*): given an arbitrary permutation π, find $d_{pref}(\pi)$, which is the minimum number of reversals of the form $\rho(1,i)$ that sort π. Their concern is with bounds on the *prefix reversal diameter* of the symmetric group, $d_{pref}(n) = \max_{\pi \in S_n} d_{pref}(\pi)$. They show that $d_{pref}(n) \leq \frac{5}{3}n + \frac{5}{3}$ (see also [GT78]) and that for infinitely many n, $d_{pref}(n) \geq \frac{17}{16}n$ [GP79]. Aigner and West [AW87] consider the diameter of sorting when the operation is reinsertion of the first element, and Amato et al. [ABSR89] consider a variation inspired by reversing trains. Kececioglu and Sankoff [KS93] found an approximation algorithm for sorting by reversals with performance guarantee 2. They also devised efficient bounds, allowing them to solve the reversal distance problem optimally or almost optimally for n ranging from 30 to 50. This range covers the biologically important case of mitochondrial genomes.

Define $d(n) = \max_{\pi \in S_n} d(\pi)$ to be the *reversal diameter* of the symmetric group of order n. Gollan conjectured that $d(n) = n - 1$ and that only one permutation γ_n, and its inverse, γ_n^{-1}, require $n - 1$ reversals to be sorted (see Kececioglu and Sankoff [KS93] for details). The *Gollan* permutation, in one-line notation, is defined as follows:

$$\gamma_n = \begin{cases} (3,1,5,2,7,4,\ldots,n-3,n-5,n-1,n-4,n,n-2), & n \text{ even}, \\ (3,1,5,2,7,4,\ldots,n-6,n-2,n-5,n,n-3,n-1), & n \text{ odd}. \end{cases}$$

For $n \leq 11$, Gollan verified this conjecture using extensive computations. Kececioglu and Sankoff [KS93] developed lower bounds for reversal distance, allowing them to verify Gollan's conjecture for $n \leq 200$ for $n \equiv 1 \pmod 3$. In the present paper, we introduce the notion of the *breakpoint graph* of a permutation and establish the links between reversal distance and *maximum cycle decomposition* of this graph. This construction allows us to prove Gollan's conjecture. Further, we study the problem of expected reversal distance between two random permutations. We demonstrate that reversal distance between two random permutations is very close to the reversal diameter of the symmetric group, thereby indicating that reversal distance provides a good separation between related and nonrelated sequences in molecular evolution studies.

Afterwards, we study reversals of *signed* permutations. The *Lobelia fervens* permutation (Fig. 1) corresponds to the signed permutation $(-7, +1, +2, +4, +5, +3, -6, +8)$. In the biologically more relevant signed case, every reversal of fragment $[i, j]$ changes the signs of the elements within that fragment. We are interested in the minimum number of reversals required to transform the signed permutation π into the identity signed permutation $(+1, +2, \ldots, +n)$. We devise an approximation algorithm

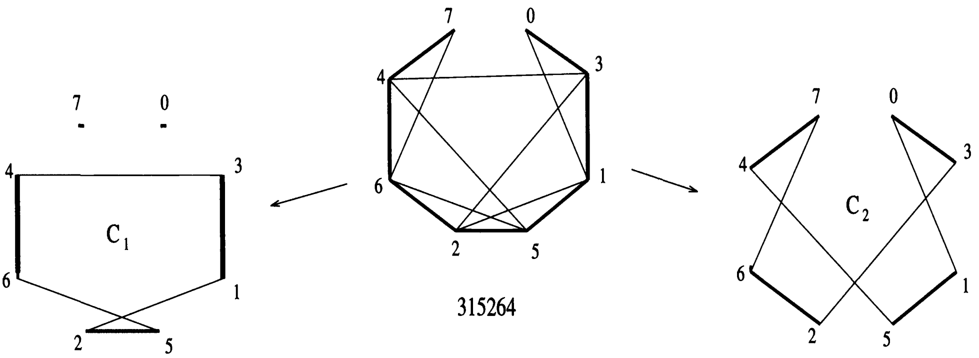


FIG. 3. *Breakpoint graph for* $\gamma_6 = 315264$ *and its maximum cycle decomposition.*

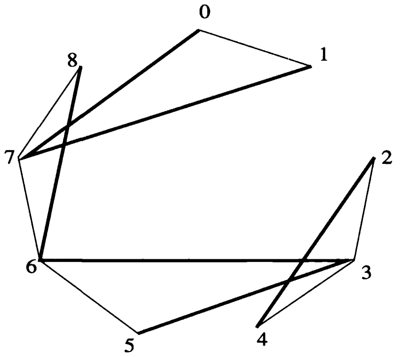


FIG. 4. *Breakpoint graph for the* Lobelia fervens *permutation.*

for sorting signed permutations by reversals with guaranteed error bound $\frac{3}{2}$. Finally, we use signed permutations to get a performance guarantee of $\frac{7}{4}$ for sorting unsigned permutations by reversals, thereby improving on the factor of 2 due to Kececioglu and Sankoff [KS93].

2. Breakpoint graph and reversal distance. Let $i \sim j$ if $|i-j| = 1$. Extend a permutation $\pi = \pi_1\pi_2\dots\pi_n$ by adding $\pi_0 = 0$ and $\pi_{n+1} = n+1$. We call a pair of consecutive elements π_i and π_{i+1}, $0 \leq i \leq n$, of π an *adjacency* if $\pi_i \sim \pi_{i+1}$, and a *breakpoint* if $\pi_i \not\sim \pi_{i+1}$. Define an edge-colored graph $G(\pi)$ with $n+2$ vertices $0, 1, \dots, n, n+1$. We join vertices i and j by a *black* edge if (i, j) is a *breakpoint* of π. We join vertices i and j by a *gray* edge if $i \sim j$ and i and j are not consecutive in π. The graph $G(\gamma_6)$, corresponding to the Gollan permutation $\gamma_6 = 315264$, is shown in Fig. 3.

A sequence of vertices $x_1x_2\dots x_m = x_1$ is called a *cycle* in a graph $G(V, E)$ if $(x_i, x_{i+1}) \epsilon E$ for $1 \leq i \leq m-1$. A cycle in an edge-colored graph G is called *alternating* if the colors of every two consecutive edges of this cycle are distinct. In the following, we consider *cycle decompositions* of $G(\pi)$ into the maximum number $c(\pi)$ of edge-disjoint alternating cycles. A maximum cycle decomposition of $G(\gamma_6)$ into 2 cycles is shown in Fig. 3. Figure 4 displays the breakpoint graph for the *Lobelia fervens* permutation, $\pi = 7124536$, with $c(\pi) = 2$. (Black edges are shown by thick lines, while gray edges are shown by thin ones.) Note that cycles can be vertex self-intersecting.

A vertex v in the graph G is called *balanced* if the number of black edges incident to v equals the number of gray edges incident to v. A *balanced graph* is a graph in which every vertex is balanced. It is easy to see that $G(\pi)$ is a balanced graph for every π; therefore, it contains an alternating Eulerian cycle in every connected component. For characterization of alternating Eulerian cycles in edge-colored graphs, see Kotzig [K68] and Pevzner [P94].

Cycle decompositions play an important role in estimating the reversal distance. When we apply a reversal to a permutation, there might be a change in the number of breakpoints, as well as in the number of cycles in a maximum decomposition. In Theorem 1, we show that there is a strong correlation between these two changes. This idea allows us to bound the reversal distance in terms of the size of the maximum cycle decomposition.

Denote the number of black edges in $G(\pi)$ (breakpoints in π) as $b = b(\pi)$ and the number of adjacencies in π as $a(\pi)$. Let $c(\pi)$ be the number of cycles in a maximum cycle decomposition of $G(\pi)$. Given an arbitrary reversal ρ, denote $\Delta b = \Delta b(\pi, \rho) = b(\pi\rho) - b(\pi)$ (increase in breakpoints), $\Delta a = \Delta a(\pi, \rho) = a(\pi\rho) - a(\pi)$ (increase in adjacencies), and $\Delta c = \Delta c(\pi, \rho) = c(\pi\rho) - c(\pi)$ (increase in the number of cycles in a maximum decomposition).

THEOREM 1. *For every permutation π and reversal ρ, $\Delta c(\pi, \rho) - \Delta b(\pi, \rho) \leq 1$.*

Proof. We augment $G(\pi)$ to get $G'(\pi)$ as follows: For every adjacency (π_i, π_{i+1}) in π, add a cycle of length 2 with one black and one gray edge connecting π_i and π_{i+1}. Every decomposition of $G(\pi)$ into $c(\pi)$ cycles corresponds to a decomposition of $G'(\pi)$ into $c(\pi) + a(\pi)$ cycles and, conversely, any decomposition of $G'(\pi)$ into $c'(\pi)$ cycles corresponds to a decomposition of $G(\pi)$ into at least $c'(\pi) - a(\pi)$ cycles.

We will prove that the number of cycles in $G'(\pi)$ changes by no more than one in a single reversal. An arbitrary reversal $\rho(i, j)$ involves 4 vertices of $G'(\pi)$ and leads to replacing two black edges *DEL*$= \{(\pi_{i-1}, \pi_i), (\pi_j, \pi_{j+1})\}$ by the black edges *ADD* $= \{(\pi_{i-1}, \pi_j), (\pi_i, \pi_{j+1})\}$.

If these two black edges in *ADD* belong to the same cycle in a maximum cycle decomposition of $G'(\pi\rho)$, then a deletion of that cycle gives a cycle decomposition of $G'(\pi)$ with at least $c'(\pi\rho) - 1$ cycles. Therefore, $c'(\pi) \geq c'(\pi\rho) - 1$ and $\Delta c' = \Delta c + \Delta a \leq 1$.

On the other hand, if the black edges in *ADD* belong to different cycles C_1 and C_2 in a maximum cycle decomposition of $G'(\pi\rho)$, then deleting $C_1 \cup C_2$ gives a set of edge-disjoint cycles of size $c'(\pi\rho) - 2$ in the graph $G(\pi\rho) \setminus (C_1 \cup C_2)$. Clearly, the set of edges $(C_1 \cup C_2 \cup DEL) \setminus ADD$ forms a balanced graph and must contain at least one cycle. Combining this cycle with the previously obtained $c'(\pi\rho) - 2$ cycles, we obtain a cycle decomposition of $G(\pi) = (G(\pi\rho) \setminus (C_1 \cup C_2)) \cup (C_1 \cup C_2 \cup DEL \setminus ADD)$ into at least $c'(\pi\rho) - 1$ cycles. Therefore, $\Delta c' = \Delta c + \Delta a \leq 1$.

Finally, observing that $a(\pi) + b(\pi) = n + 1$ implies $\Delta b + \Delta a = 0$, we prove the theorem. □

Theorem 1 immediately gives us a new lower bound for the reversal distance in terms of the number of breakpoints and the size of a maximum cycle decomposition of the breakpoint graph.

THEOREM 2. *For every permutation π, $d(\pi) \geq b(\pi) - c(\pi)$.*

Proof. Let $\rho_t, \ldots, \rho_1$ be a shortest series of reversals transforming $\pi = \pi_t$ into the identity permutation π_0. Denote $\pi_{i-1} = \pi_i \rho_i$ for $i = 1, \ldots, t$, and apply Theorem 1 for a permutation π_i and reversal ρ_i.

$$d(\pi_i) = d(\pi_{i-1}) + 1 \geq d(\pi_{i-1}) - \Delta b(\pi_i, \rho_i) + \Delta c(\pi_i, \rho_i)$$

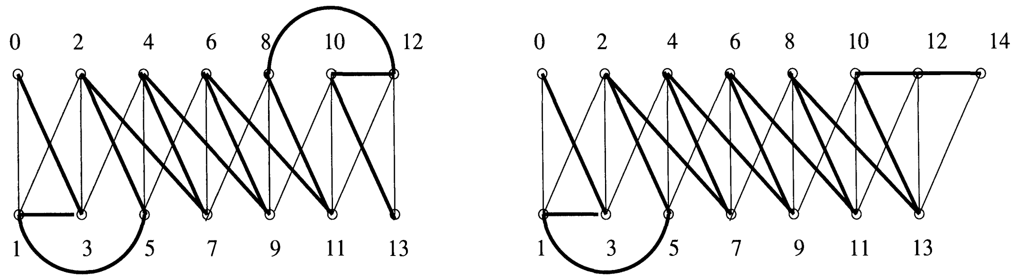


FIG. 5. $G(\gamma_{12})$ *and* $G(\gamma_{13})$.

$$= d(\pi_{i-1}) + (b(\pi_i) - b(\pi_{i-1})) + (c(\pi_{i-1}) - c(\pi_i)).$$

Recalling that $d(\pi_0) = b(\pi_0) = c(\pi_0) = 0$, we get

$$d(\pi_i)-(b(\pi_i)-c(\pi_i)) \geq d(\pi_{i-1})-(b(\pi_{i-1})-c(\pi_{i-1})) \geq \cdots \geq d(\pi_0)-(b(\pi_0)-c(\pi_0)) = 0.$$

Substituting $i = t$, we prove the theorem. □

3. Reversal diameter of the symmetric group. Now we have a characterization of the reversal distance of a permutation in terms of the maximum cycle decomposition of the breakpoint graph. Next we show that the graph corresponding to the Gollan permutation, γ_n, has at most two disjoint alternating cycles.

LEMMA 1. *Every alternating cycle in* $G(\gamma_n)$ *contains the vertex* 1 *or* 3.

Proof. Let us recall the Gollan permutation:

$$\gamma_n = \begin{cases} (3,1,5,2,7,4,\ldots,n-3,n-5,n-1,n-4,n,n-2), & n \text{ even}, \\ (3,1,5,2,7,4,\ldots,n-6,n-2,n-5,n,n-3,n-1), & n \text{ odd}. \end{cases}$$

Let i be the minimal odd vertex of an alternating cycle X in $G(\gamma_n)$. Consider the sequence i, j, k of consecutive vertices in X, where (i,j) is black, and (j,k) is gray.

If $i > 5$, then $j = i-3$ or $j = i-5$ and $k = j+1$ or $k = j-1$ (see the structure of the Gollan permutation and Fig. 5), implying that k is odd and $k < i$, a contradiction. If $i = 5$, then $j = 2$ and k is either 1 or 3, a contradiction. Therefore, i is either 1 or 3. □

THEOREM 3 (Gollan conjecture). *For every* n, $d(\gamma_n) = d(\gamma_n^{-1}) = n-1$.

Proof. For $n \leq 2$, the claim is trivial. For $n > 2$, partition the vertex set of $G(\gamma_n)$ into $V_l = \{0,1,3\}$ and V_r. From Lemma 1 and the fact that there is no cycle contained in V_l, we see that every alternating cycle must contain at least 2 edges from the cut (V_l, V_r). Because the cut (V_l, V_r) consists of 4 edges $((1,2),(1,5),(3,2),(3,4))$, the maximum number of edge-disjoint alternating cycles in a cycle decomposition of $G(\gamma_n)$ is at most $\frac{4}{2} = 2$.

From Theorem 2, $d(\gamma_n) \geq b(\gamma_n) - c(\gamma_n) \geq n+1-2 = n-1$. On the other hand, $d(\gamma_n) \leq n-1$ [WEHM82]. Finally, note that $d(\gamma_n^{-1}) = d(\gamma_n)$. □

Before we prove that γ_n and γ_n^{-1} are the only permutations in S_n with a reversal distance of $n-1$ (*strong Gollan conjecture*), we need to extend the concept of sorting permutations by reversals as follows: For any permutation of $\{1,\ldots,n\}$, $\pi = \pi_1\pi_2\ldots\pi_n$, let $\hat{\pi} = \hat{\pi}_1\hat{\pi}_2\ldots\hat{\pi}_n$, with $\hat{\pi}_i = \pi_i+1$, be a permutation of $\{2,3,\ldots,n\}$. Define $d(\hat{\pi})$ as the minimum number of reversals required to transform $\hat{\pi}$ to $234\ldots n+1$. Clearly, $d(\hat{\pi}) = d(\pi)$.

THEOREM 4 (strong Gollan conjecture). *For every n, γ_n and γ_n^{-1} are the only permutations that require $n-1$ reversals to be sorted.*

Proof. Define $\mathcal{P}_n \equiv \{\pi | \pi \in S_n \text{ and } d(\pi) = n-1\}$. We have seen that $\mathcal{P}_n \supseteq \{\gamma_n, \gamma_n^{-1}\}$. In what follows, we inductively prove that $\mathcal{P}_n \subseteq \{\gamma_n, \gamma_n^{-1}\}$.

For $n \leq 2$, the claim is trivial. Assume that the claim is true up to $n-1$. Consider $\pi \in \mathcal{P}_n$. Let ρ_π be the unique reversal that brings n to the right end, that is, $\pi \cdot \rho_\pi = \pi' n$, where π' is a permutation of $\{1, \ldots, n-1\}$.

It follows that $d(\pi) \leq 1 + d(\pi')$. Then, $d(\pi') \geq d(\pi) - 1 = n-2$. By induction, π' is either γ_{n-1} or γ_{n-1}^{-1}. Define $\mathcal{A} = \{\pi | \pi \cdot \rho_\pi = \gamma_{n-1} n\}$ and $\mathcal{B} = \{\pi | \pi \cdot \rho_\pi = \gamma_{n-1}^{-1} n\}$, where γn denotes the concatenation of a permutation γ with element n. Obviously, $\mathcal{P}_n \subseteq \mathcal{A} \cup \mathcal{B}$.

Likewise, define ρ'_π as the unique reversal that brings 1 to the left end, that is, $\pi \rho'_\pi = 1\pi'$, where π' is a permutation of $\{2, 3, \ldots, n\}$. As before, $d(\pi') \geq n-2$, which implies that π is $\hat{\gamma}_{n-1}$ or $\hat{\gamma}_{n-1}^{-1}$. Define $\mathcal{C} = \{\pi | \pi \cdot \rho'_\pi = 1\hat{\gamma}_{n-1}\}$ and $\mathcal{D} = \{\pi | \pi \cdot \rho'_\pi = 1\hat{\gamma}_{n-1}^{-1}\}$. Then, $\mathcal{P}_n \subseteq \mathcal{C} \cup \mathcal{D}$.

Therefore, $\mathcal{P}_n \subseteq \{\mathcal{A} \cup \mathcal{B}\} \cap \{\mathcal{C} \cup \mathcal{D}\}$. We state the following without proof:

1. $\mathcal{A} \cap \mathcal{C} = \phi$,
2. $\mathcal{A} \cap \mathcal{D} = \{\gamma_n\}$,
3. $\mathcal{B} \cap \mathcal{C} = \{\gamma_n^{-1}\}$,
4. $\mathcal{B} \cap \mathcal{D} = \phi$.

It follows that $\mathcal{P}_n \subseteq \{\gamma_n, \gamma_n^{-1}\}$. □

4. Expected reversal distance. For any permutation $\pi \in S_n$, consider a set of cycles that form a maximum decomposition and partition them by size. Let $c_i(\pi)$ denote the number of alternating cycles of length i in a maximum decomposition, which do not include either vertex 0 or $n+1$. There are at most two edge disjoint cycles, which contain the vertex 0 or $n+1$. Then

$$c(\pi) \leq \sum_{i=4}^{2(n+1)} c_i(\pi) + 2. \tag{1}$$

For $k \leq 2(n+1)$, let us consider cycles in the decomposition whose size is at least k. The number of such cycles is $c(\pi) - \sum_{i=4}^{k-1} c_i(\pi) - 2$. Now, the breakpoint graph of π has exactly $2b(\pi)$ edges. From this and the fact that the cycles are edge disjoint, we have

$$\forall k \leq 2(n+1), \qquad c(\pi) - \sum_{i=4}^{k-1} c_i(\pi) - 2 \leq \frac{1}{k}\left(2b(\pi) - \sum_{i=4}^{k-1} i c_i(\pi)\right). \tag{2}$$

From (1) and (2), we get

$$\forall k \leq 2(n+1), \qquad c(\pi) \leq \frac{1}{k}\left(2b(\pi) + \sum_{i=4}^{k-1}(k-i)c_i(\pi)\right) + 2. \tag{3}$$

Theorem 2 and inequality (3) imply that for all $k \leq 2(n+1)$, we can bound $d(\pi)$ as

$$d(\pi) \geq \left(1 - \frac{2}{k}\right) b(\pi) - \frac{1}{k}\left(\sum_{i=4}^{k-1}(k-i)c_i(\pi)\right) - 2 \tag{4}$$

$$\geq \left(1 - \frac{2}{k}\right) b(\pi) - \left(\sum_{i=4}^{k-1} c_i(\pi)\right) - 2. \tag{5}$$

Consider a permutation π chosen uniformly at random. As a cycle imposes a restriction on the permutation, it is intuitively clear that π does not have too many cycles. Denote the expected number of cycles of length i in a maximum cycle decomposition of $G(\pi)$ by $E(c_i(\pi)) = \frac{1}{n!}\sum_{\pi \in S_n} c_i(\pi)$. If we can bound $E(c_i(\pi))$, we can use (5) above to get a lower bound on the expected reversal distance. Lemma 2 provides such a bound which is, somewhat surprisingly, independent of n. Note that there is a slight ambiguity in the definition of $E(c_i(\pi))$, which depends on the choice of a maximum cycle decomposition for each $\pi \in S_n$. This does not affect Lemma 2, however, which holds for an arbitrary cycle decomposition.

LEMMA 2. $E(c_i(\pi)) \leq \frac{2^i}{i}$.

Proof. A cycle of length $i = 2t$ is a set of t breakpoints (unordered pairs of vertices) of the form

$$\{(x'_t, x_1), (x'_1, x_2), (x'_2, x_3), \ldots, (x'_{t-1}, x_t)\}, \text{ with } x_j \sim x'_j.$$

Consider the set $x_1, x_2, \ldots, x_t$. First, we claim that in every maximum cycle decomposition, $x_1, x_2, \ldots, x_t$ are all distinct. To see this, consider the case $x_k = x_l$, for some $1 \leq k < l \leq t$. Then $(x'_k, x_{k+1}), (x'_{k+1}, x_{k+2}), \ldots, (x'_{l-1}, x_l = x_k)$ form an alternating cycle, which can be detached to give a larger decomposition.

We have $\frac{n!}{(n-t)!}$ ways of selecting the ordered set, $x_1, x_2, \ldots, x_t$. Once this is fixed we have a choice of at most 2 elements for each of the x'_j, giving a bound of $2^t\frac{n!}{(n-t)!}$ on the number of cycles of length $2t$. Note, however, that we count each $(2t)$-cycle $2t$ times, so a tighter bound for the number of cycles of length $2t$ is $\frac{2^t}{2t}\frac{n!}{(n-t)!}$.

Choose an arbitrary $(2t)$-cycle. The number of permutations in which this cycle can occur is no more than the number of ways of permuting the remaining $n - 2t$ elements plus the t pairs that form the cycle. Additionally, each pair can be flipped to give a different order, which gives at most $2^t(n-t)!$ permutations. Let p be the probability that an arbitrary $(2t)$-cycle is present in a random permutation. Then $p \leq \frac{2^t(n-t)!}{n!}$ and

$$E(c_i(\pi) = E(c_{2t}(\pi)) \leq \sum_{\{\text{all } (2t)\text{-cycles}\}} p \leq \frac{2^{2t}}{2t} = \frac{2^i}{i}. \qquad \square$$

There are a total of $2n$ ordered adjacencies. Any such pair occurs in exactly $(n-1)!$ permutations, so the probability that it occurs in a random permutation is $\frac{1}{n}$. Then the expected number of adjacencies $E(a) = \frac{2n}{n} = 2$ and the expected number of breakpoints $E(b) = n + 1 - E(a) = n - 1$.

We use Lemma 2 and $E(b) = n - 1$ to get a bound on the expected diameter.

THEOREM 5. $E(d) \geq (1 - \frac{4.5}{\log n})n$.

Proof. From inequality (5), for all $k \leq 2(n+1)$,

$$\begin{aligned} E(d) &\geq \left(1 - \frac{2}{k}\right)E(b) - \sum_{i=4}^{k-1} E(c_i) \geq \left(1 - \frac{2}{k}\right)(n-1) - \sum_{i=4}^{k-1} 2^i/i \\ &\geq \left(1 - \frac{2}{k}\right)(n-1) - \sum_{i=4}^{k-1} 2^i \geq n - \frac{2n}{k} - \left(1 - \frac{2}{k}\right) - 2^k + 1 \\ &\geq n - \frac{2n}{k} - 2^k. \end{aligned}$$

TABLE 1
Comparison of theoretical and experimental lower bounds on expected diameter.

n	20	30	40	50	60	70	80	90	100
Theoretical	8.00	15.08	22.58	30.08	37.58	45.08	52.58	60.08	67.58
Experimental (matching)	12.60	19.92	27.34	34.05	41.09	48.73	55.80	63.33	70.49
Experimental (linear program)	12.6	20.8	28.5	35.9	43.6	51.7	58.9	67.6	74.2

Choose $k = \log \frac{n}{\log n}$. Then $2^k \leq \frac{n}{k}$ and

$$E(d) \geq \left(1 - \frac{3}{\log \frac{n}{\log n}}\right) n \geq \left(1 - \frac{4.5}{\log n}\right) n \qquad \text{for } n \geq 2^{16}.$$

Lemma 2 and inequality (4) for $k = 10$ imply that $E(d) \geq (1 - \frac{4.5}{\log n})n$ for $19 < n < 2^{16}$. For $1 \leq n \leq 19$, $(1 - \frac{4.5}{\log n})n < 1$. □

Although the bound provided by Theorem 5 is good asymptotically, it is weak for small values of n. However, the bound given by inequality (3) is tight if we select a k that gives the minimum value. We first give an example for $n = 100$ and then compare theoretical and experimental values for lower bounds on the expected diameter.

From Lemma 2 we have $E(c_4) \leq \frac{16}{4}$, $E(c_6) \leq \frac{64}{6}$, and $E(c_8) \leq \frac{128}{8}$. Consider a random permutation with $n = 100$ and $E(b) = n - 1 = 99$. Applying inequality (3) for $k = 8$, we derive $E(c) \leq \frac{1}{8}\left(198 + (8-4)4 + (8-6)\frac{64}{6}\right) + 2 \simeq 31.42$. This gives a lower bound for expected diameter with $n = 100$ as 67.58, which is close to experimental bounds based on maximum matching and linear programming [KS93]. Table 1 compares theoretical and experimental bounds over a range of n.

Kececioglu and Sankoff [KS93] use the lower bounds to prune the branch and bound tree, in order to solve the reversal distance problem. In the case of signed reversals (see below), the cycle decomposition is unique, and we can use Theorem 2 to get comparable bounds, at a significant reduction in running time.

5. Short cycles and approximation of reversal distance. Starting from this section we discuss approximation algorithms for sorting by reversals. Define a *strip* of π as an interval $[i, j]$ such that $(i-1, i)$ and $(j, j+1)$ are breakpoints and no breakpoint lies between them. A strip is *increasing* if $\pi_i < \pi_j$; otherwise it is *decreasing*. A strip of one element is both increasing and decreasing, except for π_0 and π_{n+1}, which are always increasing.

A reversal can add or remove no more than two breakpoints. Define an i-reversal, $i \in \{0, 1, 2\}$, as one that removes i breakpoints. Our task is to remove breakpoints by reversals and merge strips into one increasing strip. Clearly, we need at least $\frac{b(\pi)}{2}$ reversals. For an upper bound on the number of reversals, Kececioglu and Sankoff [KS93] give a greedy algorithm (Fig. 6) and prove the following assertions.

LEMMA 3. ([KS93]) *Let π be a permutation with a decreasing strip. Then* (i) *π has a* 1- *or* 2-*reversal, and* (ii) *if every reversal that removes a breakpoint of π leaves a permutation with no decreasing strips, then π has a* 2-*reversal.*

Partition a sequence of reversals into *rounds*, so that each round (except, perhaps, the first one) begins with a 0-reversal and has no other 0-reversals. Lemma 3 implies that in procedure *KS*, each round ends in a 2-reversal, thereby proving that every 0-reversal can be amortized against a 2-reversal and, *on the average*, we need at

```
**Procedure** $KS(\pi)$
**while** $\pi$ contains a breakpoint **do**
    $\rho = \text{Greedy}(\pi)$
    $\pi = \pi \cdot \rho$
**endwhile**

**Procedure** $Greedy(\pi)$
**begin**
    Return a reversal that removes the most breakpoints of $\pi$,
    resolving ties in favor of reversals that leave a decreasing strip.
**end**
```

FIG. 6. *The greedy algorithm.*

most one reversal to remove a breakpoint. Comparison of the upper bound of $b(\pi)$ reversals against the lower bound $\frac{b(\pi)}{2}$ provides a performance guarantee of 2. Can we do better?

Theorem 2 gives a stronger lower bound of $d(\pi) \geq b(\pi) - c(\pi)$, where $c(\pi)$ is the number of cycles in a maximum cycle decomposition of the breakpoint graph. Note that breakpoints correspond to black edges in this graph and every alternating cycle has at least 2 black edges. Therefore, the lower bound of $d(\pi) \geq \frac{b(\pi)}{2}$ is a simple corollary of Theorem 2.

Also, using inequality (2), we derive

$$d(\pi) \geq b(\pi) - c_4(\pi) - (c(\pi) - c_4(\pi)) \geq b(\pi) - c_4(\pi) - \frac{b(\pi) - 2c_4(\pi)}{3} = \frac{2}{3}b(\pi) - \frac{1}{3}c_4(\pi).$$

In the following sections we devise algorithms that sort π in at most $b(\pi) - \epsilon c_4(\pi)$ steps, for some $\epsilon > 0$. Then, the performance ratio of our algorithms is

$$(6) \qquad \mathcal{A} = \max_{0 \leq c_4(\pi) \leq \frac{b(\pi)}{2}} \left\{ \frac{b(\pi) - \epsilon c_4(\pi)}{\frac{2}{3}b(\pi) - \frac{1}{3}c_4(\pi)} \right\} \leq \begin{cases} 2 - \epsilon, & \epsilon \leq \frac{1}{2}, \\ \frac{3}{2} & \text{otherwise.} \end{cases}$$

6. Approximation algorithm for signed permutations. It is interesting to note that while the problem of sorting signed permutations is easier to handle, it is more relevant from a biological point of view. This is because genes are *directed* fragments of DNA sequences (Fig. 1).

We note that the concept of breakpoint graph as well as strips extends naturally to signed permutations. Define a transformation from a signed permutation π of order n to an unsigned permutation $\pi' \in S_{2n}$ as follows: replace $+i$ by $2i-1, 2i$ and $-i$ by $2i, 2i-1$. We observe that the identity signed permutation maps to the identity (unsigned) permutation, and the effect of a reversal on π can be mimicked by a reversal on π'. Therefore, any lower bound on π' is a lower bound on π. In particular, Theorem 2 holds.

For the upper bound, we shall perform reversals only across breakpoints so that any reversal on the unsigned permutation can be mimicked by a reversal on the signed permutation. It follows that, for our purpose, the two permutations are equivalent and in the following discussion, whenever we refer to the breakpoint graph or strips of a signed permutation, it is implied that we refer to the breakpoint graph or strips, respectively, of the transformed unsigned permutation.

Observe that in a breakpoint graph of signed permutations, every vertex has degree at most 2. Therefore the cycle decomposition is unique, thus making the case

```
Procedure SignedSort($\pi$)
 1. while $\pi$ contains a breakpoint do
 2.     if $\pi$ has no decreasing strips
 3.         if any 4-cycle $C$ remains in $G(\pi)$
 4.             Find a cycle $C'$ which crosses $C$.
 5.             Do a 0-reversal on $C'$ so that the 4-cycle $C$ is oriented.
 6.             Do a 2-reversal on the 4-cycle $C$.
 7.         else
 8.             Do a 0-reversal on an arbitrary cycle.
 9.     else
10.         $\rho = Greedy(\pi)$
11.         $\pi = \pi \cdot \rho$
12. endwhile
```

FIG. 7. *Algorithm for sorting signed permutations.*

of signed permutations easier to handle. Below, we devise an algorithm that sorts signed permutations in $b(\pi) - \frac{1}{2}c_4(\pi)$ steps, thereby achieving a performance ratio of $\frac{3}{2}$. Later, we will use signed permutations to improve the performance guarantee for (unsigned) sorting by reversals.

In order to be able to sort a signed permutation π in less than $b(\pi)$ steps, we need 2-reversals that *do not have to be amortized* against 0-reversals. In the breakpoint graph of a signed permutation, 2-reversals correspond to elimination of 4-cycles, while 1-reversals correspond to shortening of longer cycles. However, the breakpoints might be oriented in such a way that 1- and 2-reversals are infeasible.

We call $\rho(i,j)$ a *reversal on a cycle* if the breakpoints (π_{i-1}, π_i) and (π_j, π_{j+1}) belong to the *same* cycle. A cycle is *oriented* if there exists a 1- or 2-reversal on it. Two cycles are *crossing* if some of the breakpoints corresponding to their black edges are *interleaved* in the permutation. For example, for γ_6, $\rho : 315264 \rightarrow 315624$ is a reversal on the cycle $C = 134652$ (Fig. 3), since the breakpoints $(5,2)$ and $(6,4)$ belong to C. C is oriented since ρ is a 1-reversal. Cycles C_1 and C_2 in Fig. 3 are crossing as their breakpoints ($(3,1)$ and $(6,4)$ in C_1, $(1,5)$ and $(4,7)$ in C_2) are interleaved in the permutation 315264.

Note that a reversal on a cycle can orient an unoriented crossing cycle. The following lemma shows that we can use reversals on a cycle to orient 4-cycles, for signed permutations.

LEMMA 4. *Any 4-cycle C that is not oriented has a crossing cycle C'. Also, there exists a reversal on C' which will orient C.*

Proof. A 4-cycle that is not oriented is of the form $\ldots i, j \ldots j', i' \ldots$, with $i \sim i'$ and $j \sim j'$. Since (j, j') is a gray edge in the breakpoint graph, the set of elements strictly between j and j' in the permutation, $\mathcal{S}$, is nonempty. Consider the largest and the smallest (not necessarily distinct) elements in $\mathcal{S}$. For at least one of these two elements, say k, there exists $k' \sim k$ such that $k' \notin \mathcal{S}$, and $k' \neq j, j'$. Consequently, there exist breakpoints (k,l) and (k',m) which are interleaved with (i,j) and (j',i'). Also, since all cycles in signed permutations are vertex-disjoint, $k' \neq i, i'$. Then, the cycle which contains the black edges (k,l) and (k',m) and the gray edge (k,k') is a crossing cycle for the 4-cycle C, and a reversal along the edges (k,l) and (k',m) will orient C. □

Lemma 4 motivates the algorithm *SignedSort* for sorting signed permutations (Fig. 7). It uses the procedure *Greedy*, which was described earlier (Fig. 6).

LEMMA 5. *After step (6) in SignedSort, some decreasing strips remain.*

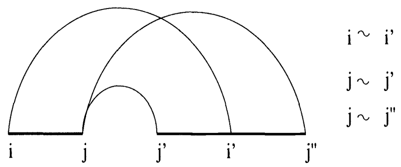


FIG. 8. *A 2-overlapping pair of cycles* (i, j, j', i') *and* (j, i, i', j'').

Proof. Step (4) is executed when all strips in π are increasing. The 0-reversal of Step (5), which is on an increasing strip, creates a decreasing strip. To remove the decreasing strip, the reversal of Step 6 would need to be on the same interval, which it is not. □

LEMMA 6. *If there is a* 4*-cycle in* $G(\pi)$ *at the beginning of any round of SignedSort (except, perhaps, the first one), then there are at least two* 2*-reversals in that round.*

Proof. If there is a 4-cycle in $G(\pi)$ at the beginning of any round (except, perhaps, the first one), that round begins with a 0-reversal followed by a 2-reversal. (Note that this cycle is nonoriented since all cycles in a permutation with no decreasing strips are nonoriented.) Also, from Lemma 5, some decreasing strips remain after this 2-reversal. At the point, as the permutation has decreasing strips we call *Greedy.* Then Lemma 3 implies that every round of *Greedy* ends in a 2-reversal, thus proving that as long as there are 4-cycles at the beginning of a round in *SignedSort*, there are at least two 2-reversals in that round. □

THEOREM 6. *SignedSort sorts a signed permutation* π *in at most* $b(\pi) - \frac{1}{2}c_4(\pi)$ *reversals and provides an approximation ratio of* $\frac{3}{2}$.

Proof. Let d_i be the number of i-reversals in *SignedSort.* Observing that $b(\pi) = d_1 + 2d_2$, *SignedSort* sorts π in $d_0 + d_1 + d_2 = b(\pi) + d_0 - d_2$ steps.

Each round of *SignedSort* has exactly one 0-reversal (except, perhaps, the first one) and at least one 2-reversal, implying $d_2 \geq d_0$. In addition, by Lemma 6, if there is a 4-cycle in π at the beginning of a round, then there are at least two 2-reversals in that round. Therefore, if there are r such rounds, then $d_2 \geq d_0 + r$. On the other hand, each of the 4-cycles in $c_4(\pi)$ must be removed in a 2-reversal in one of these r rounds (4-cycles in signed permutations are unaffected by the other reversals). These $c_4(\pi)$ 2-reversals in r rounds together with 2-reversals in each of the remaining $d_0 - r$ rounds imply that $d_2 \geq d_0 + c_4(\pi) - r$.

From these two inequalities, we have $d_2 \geq d_0 + \frac{1}{2}c_4(\pi)$, which implies that *SignedSort* sorts π in $b(\pi) + d_0 - d_2 \leq b(\pi) - \frac{1}{2}c_4(\pi)$ steps. The bound on performance follows from equation (6). □

7. Approximation algorithm for sorting by reversals.

In general, finding a maximum cycle decomposition is not straightforward. In this section, we concentrate only on finding a cycle decomposition (not necessarily maximum) with a large number of 4-cycles, since such a decomposition will provide an improved performance ratio for sorting by reversals.

Any two 4-cycles can share at most two edges. Two 4-cycles are 2*-overlapping* if they share two edges.

LEMMA 7. *If two* 4*-cycles of* $G(\pi)$ *are* 2*-overlapping, then one of them is oriented (i.e., a* 2*-reversal is possible on it).*

Proof. The only way a pair of alternating 4-cycles can 2-overlap is shown in Fig. 8. It can be verified that in any permutation of i, j, j', i', j'' that preserves such a structure, one of the 4-cycles is oriented. □

The 4*-cycle graph* $H(\pi)$ of a permutation π is defined as a graph in which each node corresponds to a 4-cycle of $G(\pi)$, and two nodes are connected by an edge if the corresponding 4-cycles share an edge in $G(\pi)$. An *independent set* in $H(\pi)$ corresponds to a set of edge disjoint 4-cycles. Furthermore, this graph has bounded degree, and there are efficient algorithms to find good approximations to a maximum independent set in bounded degree graphs.

We call a graph *strongly d-bounded* if the degree of every vertex in the graph is bounded by d and the degree of at least one vertex in every connected component of the graph is less than d.

LEMMA 8. *If $G(\pi)$ has no 2-overlapping cycles, then the 4-cycle graph $H(\pi)$ is strongly 4-bounded.*

Proof. It is easy to see that in a graph $G(\pi)$ without 2-overlapping cycles, a 4-cycle can have at most four other alternating cycles sharing edges with it. Consider a 4-cycle C containing the maximal element i of π among all 4-cycles in a given connected component of $H(\pi)$. Since $i+1$ does not belong to any 4-cycle in this component, at most one of the neighboring cycles of C contains the vertex of $G(\pi)$ corresponding to that maximal element. It follows that C has at most three neighbors in $H(\pi)$. □

LEMMA 9. *In a strongly d-bounded graph $G(V,E)$, an independent set of size at least $\frac{|V|}{d}$ can be computed in $O(E)$ time.*

Proof. Pick a vertex of degree less than d, add it to the independent set, and remove the neighbors. Clearly, the remaining graph is strongly d-bounded. Repeat this step until no vertices are left. As no more than d vertices are removed in each step, the size of the independent set is at least $\frac{|V|}{d}$. □

LEMMA 10. *In a strongly d-bounded graph $G(V,E)$, a $\frac{2}{(d+1)}$ approximation to a maximum independent set can be computed in $O(E)$ time.*

Proof. Let the size of the maximum independent set be $a \cdot n$, for some a, where $|V| = n$. Consequently, the minimum vertex cover for G has a size $= (1-a)\cdot n$. We can find a vertex cover V' of size at most $\min\{2(1-a)\cdot n, n\}$ in $O(E)$ time [CLR90]. Then $I_1 = V \backslash V'$ is an independent set for G, of size at least $\max\{(2a-1)\cdot n, 0\}$. Another independent set I_2 is given by Lemma 9 and is of size $\frac{n}{d}$. Obviously, we select the larger of the two sets.

There are two cases in analyzing the performance of our approximation:

$\max\{(2a-1)n, 0\} = 0$:

This implies that $a \leq \frac{1}{2}$. In this case we select I_2. Performance is $\frac{1}{ad} \geq \frac{2}{d}$.

$\max\{(2a-1)n, 0\} = (2a-1)n$:

In this case we select the larger of the two sets, I_1 and I_2. The performance is $\max\{\frac{1}{da}, \frac{(2a-1)}{a}\}$, with $a > \frac{1}{2}$. In the worst case, $\frac{1}{ad} = \frac{(2a-1)}{a}$, which implies that $a = \frac{(d+1)}{2d}$, and performance is $\frac{2}{(d+1)}$. □

Lemmas 7, 8, and 10 motivate the algorithm *ReversalSort* (Fig. 9). Theorem 7 provides a performance guarantee for this algorithm.

THEOREM 7. *The algorithm ReversalSort achieves an approximation ratio of $\frac{9}{5}$.*

Proof. Note that every reversal on the signed permutation σ' in step (4) of *ReversalSort* can be simulated on the unsigned permutation σ. Therefore, an upper bound on the number of reversals in *SignedSort*(σ') is an upper bound on the number of reversals in step (4) of *ReversalSort.* If the number of 4-cycles found in the cycle decomposition in step (2) is $c_4'(\sigma)$, then from Theorem 6, we have the bound $d(\sigma) \leq b(\sigma) - \frac{1}{2}c_4'(\sigma)$ for the number of reversals in step (4) of *ReversalSort.* Let $\rho_1\rho_2\ldots\rho_x$ be 2-reversals chosen in step (1) of the algorithm, so that $\pi\rho_1\rho_2\ldots\rho_x = \sigma$. Clearly, $b(\pi) - b(\sigma) = 2x$. Therefore, *ReversalSort* requires no more than $x + b(\sigma) - \frac{1}{2}c_4'(\sigma) = b(\pi) - x - \frac{1}{2}c_4'(\sigma)$ reversals to sort π.

```
Algorithm ReversalSort(π)
  1. Starting with the permutation $\pi$, perform 2-reversals on $G(\pi)$ until no 2-overlapping
     cycles remain (Lemma 7). Let $\sigma$ denote the resulting permutation.
  2. Use the Independent Set approximation in $H(\sigma)$ to find a set of nonoverlapping
     4-cycles of size at least $\frac{2}{5}c_4(\sigma)$ in the breakpoint graph $G(\sigma)$ (Lemmas 8 and 10).
     Find an arbitrary cycle decomposition of the remaining edges.
  3. Split vertices of degree 4 in $G(\sigma)$ according to the cycle decomposition found in step
     (2), so that the cycles are vertex disjoint. In terms of strips, replace single elements
     by strips, oriented appropriately, resulting in a signed permutation $\sigma'$.
  4. Call SignedSort($\sigma'$) to sort $\sigma'$. Sorting of $\sigma'$ mimics sorting of $\sigma$.
```

FIG. 9. *Algorithm for sorting by reversals.*

Let $i(\delta)$ be the size of the maximum independent set in a 4-cycle graph $H(\delta)$. Lemmas 8 and 10 guarantee that $c'_4(\sigma) \geq \frac{2}{5}i(\sigma)$. On the other hand, $i(\pi) - i(\sigma) \leq 4x$, since every reversal in step (1) "destroys" at most four of the $i(\pi)$ vertices of the maximum independent set in $H(\pi)$ (there are at most 4 nonoverlapping cycles sharing edges with a 4-cycle). Therefore,

$$\begin{aligned} d(\pi) &\leq b(\pi) - x - \tfrac{1}{5}i(\sigma) &&\leq b(\pi) - x - \tfrac{1}{5}(i(\pi) - 4x) \\ &\leq b(\pi) - \tfrac{1}{5}x - \tfrac{1}{5}c_4(\pi) &&\leq b(\pi) - \tfrac{1}{5}c_4(\pi). \end{aligned}$$

The bound on performance follows from equation (6). □

8. Improved approximation for sorting by reversals. In this section, we modify *ReversalSort* to improve the performance ratio. Recall that in step (1) of *ReversalSort*(π), we perform 2-reversals to transform π into a permutation σ, which has the property that $H(\sigma)$ is strongly 4-bounded. This allows us to find a set of 4-cycles of size at least $\frac{2}{5}c_4(\sigma)$. In this section, we transform π using 2-reversals into a permutation σ, so that $H(\sigma)$ is *bipartite*. Consequently, we can find a maximum set of nonoverlapping 4-cycles in σ, which leads to improved performance.

The problem of transforming π into ν is equivalent to sorting $\nu^{-1}\pi$, and, for convenience, we shall switch between the two notations. Denote $G(\pi, \nu) \equiv G(\nu^{-1}\pi)$, $b(\pi, \nu) \equiv b(\nu^{-1}\pi)$, $i(\pi, \nu) \equiv i(\nu^{-1}\pi)$. Observe that $G(\pi, \nu)$ and $G(\nu, \pi)$ coincide, but have reversed colors, i.e., black (gray) edges in $G(\pi, \nu)$ are gray (black) in $G(\nu, \pi)$. It follows that $b(\pi, \nu) = b(\nu, \pi)$ and $i(\pi, \nu) = i(\nu, \pi)$.

Figure 10 describes the procedure *Transform* that transforms π into a permutation σ with the following properties.

LEMMA 11. *$H(\sigma)$ is bipartite.*

Proof. Consider a 4-cycle in $G(\sigma)$ formed by the vertices $\sigma_i, \sigma_{i+1}, \sigma_j, \sigma_{j+1}$, with $i + 1 < j$. Because no 2-reversals are possible on $\sigma = Transform(\pi)$, every 4-cycle in $G(\sigma)$ must be nonoriented. Therefore, we must have $\sigma_i \sim \sigma_{j+1}$ and $\sigma_j \sim \sigma_{i+1}$. Now, observe that these vertices form a 4-cycle in $G(\sigma^{-1})$ also, with the color on the edges reversed. If $\sigma_j - \sigma_{i+1} = \sigma_{j+1} - \sigma_i$, then the 4-cycle in $G(\sigma^{-1})$ is oriented. Therefore, $\sigma_j - \sigma_{i+1} = -(\sigma_{j+1} - \sigma_i)$.

To interpret this graphically, direct black edges of $G(\sigma)$ from π_k to π_{k+1} and gray edges from k to $k + 1$. We call edges of a cycle *coordinated* if they are directed the same way along this cycle. Note that if black edges of a 4-cycle are not coordinated, then this cycle is oriented. Since there is no oriented cycles in $G(\sigma)$, black edges in every 4-cycle of $G(\sigma)$ are coordinated. Also, since there is no oriented cycles in $G(\sigma^{-1})$, black edges in every 4-cycle of $G(\sigma^{-1})$ are coordinated. An observation that

```
Procedure Transform($\pi$)
begin
    $\bar{\pi} = \pi$
    $\bar{\nu} = \imath$        /* $\imath$ is the identity permutation*/
    while $\bar{\nu}^{-1} \cdot \bar{\pi}$ or $\bar{\pi}^{-1} \cdot \bar{\nu}$ has a 2-reversal, $\rho$
        if $\rho$ is a 2-reversal on $\bar{\nu}^{-1} \cdot \bar{\pi}$
            $\bar{\pi} = \bar{\pi} \cdot \rho$
        else $\bar{\nu} = \bar{\nu} \cdot \rho$
    endwhile
    return $\bar{\nu}^{-1} \cdot \bar{\pi}$
end
```

FIG. 10. *Algorithm for preprocessing permutation π using 2-reversals.*

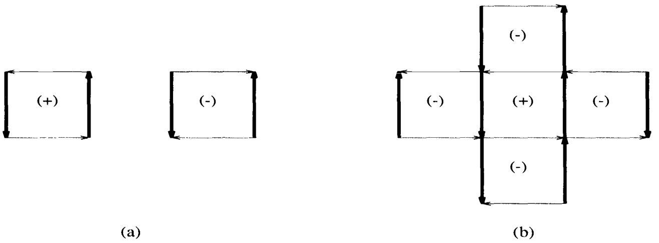


FIG. 11. (+) *and* (−) *cycles in $G(\pi)$.*

the edges of $G(\sigma)$ and $G(\sigma^{-1})$ have the same directions but reverse colors implies that gray edges in every 4-cycle of $G(\sigma)$ are coordinated. Only two such 4-cycles (denoted (+) and (−)) are possible (Fig. 11a). Further, observe that all cycles that share an edge with a cycle of type (+), must be of type (−), and vice-versa (Fig. 11b). This implies that $H(\sigma)$ is bipartite. □

LEMMA 12. *σ can be sorted in $b(\sigma) - \frac{1}{2}i(\sigma)$ reversals.*

Proof. Because $H(\sigma)$ is a bipartite graph (Lemma 11), we can reduce maximum independent set problem in $H(\sigma)$ to the maximum matching problem [CLR90]. Moreover, since $H(\sigma)$ is a graph of bounded degree, we can find a maximum independent set in $H(\sigma)$ in $O(n^{\frac{3}{2}})$ time by the maximum matching algorithm [HK73]. This implies that we can find a cycle decomposition of G(σ) in which the number of 4-cycles is $i(\sigma)$. The lemma then follows from Theorem 6. □

Let $\rho_1\rho_2\ldots\rho_x$ be the sequence of reversals in *Transform* that transforms π into $\bar{\pi}$ ($\pi\rho_1\rho_2\ldots\rho_x = \bar{\pi}$) and $\varrho_1\varrho_2\ldots\varrho_y$ be the sequence of reversals that transforms $\imath$ into $\bar{\nu}$ ($\imath\varrho_1\varrho_2\ldots\varrho_y = \bar{\nu}$).

LEMMA 13. *Let $\sigma =$ Transform(π). Then, $b(\pi) - b(\sigma) = 2(x+y)$, and $i(\pi) - i(\sigma) \leq 4(x+y)$.*

Proof. In each iteration of the **while** loop in *Transform*, $(\bar{\pi}, \bar{\nu})$ is transformed into $(\bar{\pi}', \bar{\nu}')$, in one of two ways. Either (i) $\bar{\pi}' = \bar{\pi} \cdot \rho$, $\bar{\nu}' = \bar{\nu}$, or (ii) $\bar{\pi}' = \bar{\pi}$, $\bar{\nu}' = \bar{\nu} \cdot \rho$. In case (ii), ρ is a 2-reversal on $\bar{\pi}^{-1}\bar{\nu}$, which implies that $2 = b(\bar{\pi}^{-1}\bar{\nu}) - b((\bar{\pi}')^{-1}\bar{\nu}') = b(\bar{\nu}, \bar{\pi}) - b(\bar{\nu}', \bar{\pi}') = b(\bar{\pi}, \bar{\nu}) - b(\bar{\pi}', \bar{\nu}')$. A similar argument holds for case (i), implying that $b(\bar{\pi}, \bar{\nu}) - b(\bar{\pi}', \bar{\nu}') = 2$. $i(\bar{\pi}, \bar{\nu}) - i(\bar{\pi}', \bar{\nu}') \leq 4$ follows from the fact that every 2-reversal can destroy at most four nonoverlapping 4-cycles in the breakpoint graph.

The lemma follows from the fact that each reversal in *Transform* belongs to either the sequence $\rho_1\rho_2\ldots\rho_x$ or the sequence $\varrho_1\varrho_2\ldots\varrho_y$, implying a total of $x+y$ reversals. □

ImprovedSort (Fig. 12) exploits the structure of the permutation $\sigma =$ *Trans-*

```
Algorithm ImprovedSort($\pi$)
  1. Call Transform($\pi$) to find a sequence of reversals $\rho_1\rho_2\dots\rho_x$ and $\varrho_1\varrho_2\dots\varrho_y$ such that $\pi\rho_1\rho_2\dots\rho_x = \bar{\pi}$ and $\iota\varrho_1\varrho_2\dots\varrho_y = \bar{\nu}$. Let $\sigma = \bar{\nu}^{-1}\bar{\pi}$.
  2. Find a maximum set of nonoverlapping 4-cycles in $G(\sigma)$ by solving the maximum independent set problem in the bipartite graph $H(\sigma)$ (Lemma 11). Find an arbitrary cycle decomposition of the remaining edges.
  3. Split vertices of degree 4 in $G(\sigma)$ according to the cycle decomposition found in step (2), so that the cycles are vertex disjoint. In terms of strips, replace single elements by strips, oriented appropriately, resulting in a signed permutation $\sigma'$.
  4. Call SignedSort($\sigma'$) to find a sequence of reversals that sort $\sigma'$. Note that this sequence is mimicked by a sequence of reversals $\varphi_1\varphi_2\dots\varphi_z$ that sorts $\sigma$.
  5. Apply the sequence of reversals $\rho_1\rho_2\dots\rho_x\varphi_1\varphi_2\dots\varphi_z\varrho_y\dots\varrho_2\varrho_1$ to sort $\pi$.
```

FIG. 12. *Improved algorithm for sorting by reversals.*

form(π) to sort π more efficiently. Theorem 8 analyzes the performance of this improved algorithm.

THEOREM 8. *ImprovedSort sorts π in $b(\pi) - \frac{1}{4}c_4(\pi)$ steps.*

Proof. From step (5) of *ImprovedSort*, $d(\pi) \leq x + y + z$. Lemmas 12 and 1 3 and the inequality $i(\pi) \geq c_4(\pi)$ provide an upper bound

$$\begin{aligned} x + y + z &\leq x + y + b(\sigma) - \tfrac{1}{2}i(\sigma) \\ &= b(\pi) - (x + y) - \tfrac{1}{2}i(\sigma) \\ &\leq b(\pi) - \tfrac{(i(\pi) - i(\sigma))}{4} - \tfrac{1}{2}i(\sigma) \\ &\leq b(\pi) - \tfrac{1}{4}i(\pi) \\ &\leq b(\pi) - \tfrac{1}{4}c_4(\pi). \qquad \square \end{aligned}$$

Theorem 8 and equation (6) imply Corollary 1.

COROLLARY 1. *The algorithm ImprovedSort achieves an approximation ratio of* $\frac{7}{4}$.

9. Running time. We show that all our algorithms have complexity of $O(n^2)$. Consider lines 4–6 of *SignedSort*. If we maintain both π and π^{-1}, it takes $O(n)$ time to find a 4-cycle, as well as a crossing cycle. Kececioglu and Sankoff [KS93] give an $O(n)$ implementation of *Greedy*. Finally, no more than $n - 1$ reversals are required to sort a permutation π, which gives an upper bound of $O(n^2)$ for *SignedSort*. For *ReversalSort* and *ImprovedSort*, the most expensive preprocessing step is that of finding the maximum independent set in a bipartite graph, which takes $O(n^{\frac{3}{2}})$ time. Therefore, the time bounds do not change for the improved sorting algorithms.

10. Genome rearrangements. Sequence alignment is often the first step in molecular evolution studies. However, in many cases sequence alignment is very unreliable, thus making further evolutionary tree reconstruction almost impossible. For example, the similarity between many genes in herpes viruses is so low that it is frequently indistinguishable from the background noise. As a result, the classical methods of *sequence comparison* frequently lead to ambiguous conclusions for such highly diverged genomes, and alternative methods are sought (Karlin et al. [KMS94]). Since it is often found that the order of genes is much more conserved than the DNA sequence, an approach based on comparison of *gene orders* (Sankoff [S93]) versus traditional comparison of *DNA sequences* seems to be a method of choice for many "hard-to-analyze" genomes.

Surprisingly enough, genome comparison might also outperform gene comparison for such highly conserved genomes as plant mitochondrial DNA(mtDNA). The point mutation rate in plant mtDNA is estimated to be 100 times slower than in animal mtDNA, and many genes are 99%–99.9% identical in related species (Palmer and Herbon [PH88]). However, although there is little change in the DNA sequence of the plant mitochondrial genes from one species to another, there is rapid and extensive change in gene arrangements. This implies that the traditional methods for comparing sequences are not very conclusive in this case either.

The described algorithms for sorting by reversals were implemented and applied for analysis of genome rearrangements in herpes viruses, plant organelles, and mammalian X chromosomes. For plant organelles, Bafna and Pevzner [BP95] corrected previously postulated scenarios for genome rearrangements in the *Brassica* family. For herpes viruses, Hannenhalli et al. [HCKP95] were able to come out with three alternative gene orders in common herpesvirus ancestor. Surprisingly enough, in all biological examples we analyzed, $d(\pi) = b(\pi) - c(\pi)$, thus indicating that the bound provided by Theorem 2 might be rather tight. A similar observation has been made by Kececiougli and Sankoff [KS94]: $b(\pi) - c(\pi)$ approximated $d(\pi)$ with accuracy 1 for each of 100 randomly chosen permutations on 10,000 elements. It indicates that the bound provided by the Theorem 2 is extremely tight and raises the problem of finding an exact estimate for reversal distance in terms of cycle decompositions.

Acknowledgments. We would like to thank John Kececioglu and David Sankoff for sending us the manuscript of their paper and for useful discussions. We are grateful to Piotr Berman for useful comments on approximating maximum independent sets of bounded degree graphs, and to Sing-Hoi Sze for carefully reading the manuscript and pointing out errors. We also thank both referees for many valuable comments.

REFERENCES

[AW87] M. Aigner and D. B. West, *Sorting by insertion of leading element*, J. Combin. Theory Ser. A, 45 (1987), pp. 306–309.

[ABSR89] N. Amato, M. Blum, S. Irani, and R. Rubinfeld, *Reversing trains: A turn of the century sorting problem*, J. Algorithms, 10 (1989), pp. 413–428.

[BP95] V. Bafna and P. Pevzner, *Sorting by reversals: Genome rearrangements in plant organelles and evolutionary history of X chromosome*, Molec. Biol. Evolution, 12 (1995), pp. 239–246.

[CLR90] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, MIT Press, Cambridge, MA, 1990.

[EG81] S. Even and O. Goldreich, *The minimum-length generator sequence problem is NP-hard*, J. Algorithms, 2 (1981), pp. 311–313.

[FH89] C. Fauron and M. Havlik, *The maize mitochondrial genome of the normal type and the cytoplasmic male sterile type T have very different organization*, Current Genetics, 15 (1989), pp. 149–154.

[GP79] W. H. Gates and C. H. Papadimitriou, *Bounds for sorting by prefix reversals*, Discrete Math., 27 (1979), pp. 47–57.

[GT78] E. Gyori and E. Turan, *Stack of pancakes*, Studia Sci. Math. Hungar., 13 (1978), pp. 133–137.

[HCKP95] S. Hannenhalli, C. Chappey, E. Koonin, and P. Pevzner, *Genome sequence comparison and scenarios for genome rearrangement: A test case*, Genomics, (1995).

[HBB92] R. J. Hoffmann, J. L. Boore, and W. M. Brown, *A novel mitochondrial genome organization for the blue mussel*, Mytilus edulis, Genetics, 131 (1992), pp. 397–412.

[HK73] J. E. Hopcroft and R. Karp, *An $n^{5/2}$ algorithm for maximum matching in bipartite graphs*, SIAM J. Comput., 2 (1973), pp. 225–231.

[HP94] S. B. Hoot and J. D. Palmer, *Structural rearrangements including parallel inversions within the chloroplast genome of anemone and related genera*, J. Molec. Evolution, 38 (1994), pp. 274–281.

[J85] M. R. JERRUM, *The complexity of finding minimum-length generator sequences*, Theoret. Comput. Sci., 36 (1985), pp. 265–289.

[KMS94] S. KARLIN, E. S. MOCARSKI, AND G. A. SCHACHTEL, *Molecular evolution of herpesviruses: Genomic and protein sequence comparisons*, J. Virology, 68 (1994), pp. 1886–1902.

[KS93] J. KECECIOGLU AND D. SANKOFF, *Exact and approximation algorithms for the reversal distance between two permutations*, Proc. 4th Annual Symposium on Combinatorial Pattern Matching, A. Apostolico, M. Crochemore, Z. Galil, and U. Manber, eds., Padova, Italy, 1993; Lecture Notes in Comput. Sci., 684 (1993), pp. 87–105; Algorithmica, 13 (1995), pp. 180–210.

[KS94] ———, *Efficient bounds for oriented chromosome inversion distance*, Proc. 5th Annual Symposium on Combinatorial Pattern Matching, M. Crochemore and D. Gusfield, eds., Asilomar, CA, 1994; Lecture Notes in Comput. Sci., 807 (1994), pp. 307–325.

[KDP93] E. B. KNOX, S. R. DOWNIE, AND J. D. PALMER, *Chloroplast genome rearrangements and evolution of giant lobelias from herbaceous ancestors*, Molec. Biol. Evolution, 10 (1993), pp. 414–430.

[K93] E. V. KOONIN AND V. V. DOLJA, *Evolution and taxonomy of positive-strand RNA viruses: Implications of comparative analysis of amino acid sequences*, Crit. Rev. Biochem. Molec. Biol., 28 (1993), pp. 375–430.

[K68] A. KOTZIG, *Moves without forbidden transitions in a graph*, Mat. Casopis, 18 (1968), pp. 76–80.

[L88] M. F. LYON, *X-chromosome inactivation and the location and expression of X-linked genes*, Amer. J. Human Genetics, 42 (1988), pp. 8–16.

[PH87] J. D. PALMER AND L. HERBON, *Unicircular structure of the* Brassica hirta *mitochondrial genome*, Current Genetics, 11 (1987), pp. 565–570.

[PH88] ———, *Plant mitochondrial DNA evolves rapidly in structure, but slowly in sequence*, J. Molec. Evolution, 27 (1988), pp. 65–74.

[P94] P. PEVZNER, *DNA physical mapping and alternating Eulerian cycles in colored graphs*, Algorithmica, 13 (1995), pp. 77–105.

[RJ92] L. A. RAUBESON AND R. K. JANSEN, *Chloroplast DNA evidence on the ancient evolutionary split in vascular land plants*, Science, 255 (1992), pp. 1697–1699.

[SLA92] D. SANKOFF, G. LEDUC, N. ANTOINE, B. PAQUIN, B. F. LANG, AND R. CEDERGREN, *Gene order comparisons for phylogenetic inference: Evolution of the mitochondrial genome*, Proc. Natl. Acad. Sci. U.S.A., 89 (1992), pp. 6575–6579.

[S93] D. SANKOFF, *Analytical approaches to genomic evolution*, Biochimie, 75 (1993), pp. 409–413.

[WPFJ89] J. WHITING, M. PLILEY, J. FARMER, AND D. JEFFERY, *In situ hybridization analysis of chromosomal homologies in* Drosophila melanogaster *and* Drosophila virilis, Genetics, 122 (1989), pp. 99–109.

[WEHM82] G. A. WATTERSON, W. J. EWENS, T. E. HALL, AND A. MORGAN, *The chromosome inversion problem*, J. Theoret. Biol., 99 (1982), pp. 1–7.

SIAM J. COMPUT.
Vol. 25, No. 2, p. 290, April 1996

004

NOTE ON "A LINEAR-TIME ALGORITHM FOR COMPUTING *K*-TERMINAL RELIABILITY IN A SERIES-PARALLEL NETWORK"*

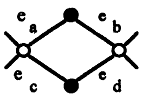
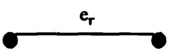
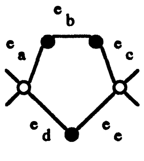
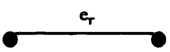
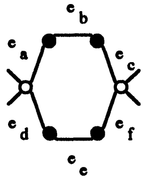
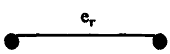
A. SATYANARAYANA†, R. K. WOOD‡, L. CAMARINOPOULOS§, AND G. PAMPOUKIS§

In an original and very interesting paper (Satyanarayana and Wood [1]) concerning polygon-to-chain reductions in a stochastic network, a small inconsistency occurs in the proof of Theorem 1. In particular, this happens in cases where the whole set of $\boldsymbol{K}$-vertices lies in the remaining polygon. Such an example is the case of polygon type 5, where the appropriate note has been made by the authors for $|\boldsymbol{K}| = 2$. Analogous notes must also be made for polygon types 4, 6, and 7, when $\boldsymbol{K}= 2$, 3, and 4, respectively. The correct transformations are given in Table 1 (note that dark vertices are $\boldsymbol{K}$-vertices).

Key words. algorithms, complexity, network reliability, series-parallel graphs, reliability-preserving reductions

AMS subject classifications. 62N05, 90B12, 60C05, 05A10

TABLE 1

polygon type	chain type	reduction formulas	new edge probability
 (4)		$r_1 = p_a q_b q_c p_d + q_a p_b p_c q_d$ $r_2 = q_a p_b q_c p_d + p_a p_b q_c q_d$ $+ p_a p_b p_c p_d \left(1 + \frac{q_a}{p_a} + \frac{q_b}{p_b} + \frac{q_c}{p_c} + \frac{q_d}{p_d}\right)$	$p_r = \frac{r_2}{r_1 + r_2}$ $\Omega = r_1 + r_2$
 (6)		$r_1 = p_a q_b p_c (p_d q_e + q_d p_e)$ $+ p_b (q_a p_c p_d q_e + p_a q_c q_d p_e)$ $r_2 = q_a p_b p_c q_d p_e + p_a p_b q_c p_d q_e$ $+ p_a p_b p_c p_d p_e \left(1 + \frac{q_a}{p_a} + \frac{q_b}{p_b} + \frac{q_c}{p_c} + \frac{q_d}{p_d} + \frac{q_e}{p_e}\right)$	$p_r = \frac{r_2}{r_1 + r_2}$ $\Omega = r_1 + r_2$
 (7)		$r_1 = p_a q_b p_c (q_d p_e p_f + p_d q_e p_f + p_d p_e q_f)$ $+ p_a p_b q_c p_f (p_d q_e + q_d p_e)$ $+ q_a p_b p_c p_d (q_e p_f + p_e q_f)$ $r_2 = q_a p_b p_c q_d p_e p_f + p_a p_b q_c p_d p_e q_f$ $+ p_a p_b p_c p_d p_e p_f \left(1 + \frac{q_a}{p_a} + \frac{q_b}{p_b} + \frac{q_c}{p_c} + \frac{q_d}{p_d} + \frac{q_e}{p_e} + \frac{q_f}{p_f}\right)$	$p_r = \frac{r_2}{r_1 + r_2}$ $\Omega = r_1 + r_2$

REFERENCE

[1] A. SATYANARAYANA AND R. K. WOOD, *A linear-time algorithm for computing* $\boldsymbol{K}$*-terminal reliability in a series-parallel network*, SIAM J. Comput., 14 (1985), pp. 818–832.

* Received by the editors October 29, 1993; accepted for publication (in revised form) August 25, 1994.

† Department of Pure and Applied Mathematics, Stevens Institute of Technology, Hoboken, NJ 07030.

‡ Department of Operations Research, Naval Postgraduate School, Monterey, CA 93943.

§ School of Engineering, Aristotle University of Thessaloniki, 540 06 Thessaloniki, Greece.

SIAM J. COMPUT.
Vol. 25, No. 2, pp. 291–311, April 1996

005

UPWARD PLANAR DRAWING OF SINGLE-SOURCE ACYCLIC DIGRAPHS*

MICHAEL D. HUTTON[†] AND ANNA LUBIW[‡]

Abstract. An upward plane drawing of a directed acyclic graph is a plane drawing of the digraph in which each directed edge is represented as a curve monotone increasing in the vertical direction. Thomassen has given a nonalgorithmic, graph-theoretic characterization of those directed graphs with a single source that admit an upward plane drawing. This paper presents an efficient algorithm to test whether a given single-source acyclic digraph has an upward plane drawing and, if so, to find a representation of one such drawing. This result is made more significant in light of the recent proof by Garg and Tamassia that the problem is NP-complete for general digraphs.

The algorithm decomposes the digraph into biconnected and triconnected components and defines conditions for merging the components into an upward plane drawing of the original digraph. To handle the triconnected components, we provide a linear algorithm to test whether a given plane drawing of a single-source digraph admits an upward plane drawing with the same faces and outer face, which also gives a simpler, algorithmic proof of Thomassen's result. The entire testing algorithm (for general single-source directed acyclic graphs) operates in $O(n^2)$ time and $O(n)$ space (n being the number of vertices in the input digraph) and represents the first polynomial-time solution to the problem.

Key words. algorithms, upward planar, graph drawing, graph embedding, graph decomposition, graph recognition, planar graph, directed graph

AMS subject classifications. 68Q20, 68Q25, 68R05, 68R10

1. Introduction.

There is a wide range of results dealing with drawing, representing, or testing planarity of graphs [5]. Steinitz and Rademacher [22], Fáry [10], Stein [21], and Wagner [26] independently showed that every planar graph can be drawn in the plane using only straight line segments for the edges. Tutte [25] showed that every 3-connected planar graph admits a convex straight-line drawing, where the facial cycles other than the unbounded face are all convex polygons. The first linear-time algorithm for testing planarity of a graph was given by Hopcroft and Tarjan [14].

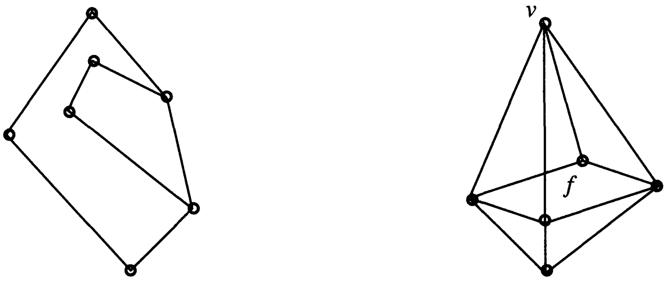
Planar graph layout has many interesting applications and has been widely studied as a method to visualize structures commonly modeled as graphs [5]. Combinational boolean circuits, subroutine call-charts, PERT graphs, isa-hierarchies in artificial intelligence, and many other objects are naturally described with directed acyclic graphs and are best understood visually when all edges are drawn in the same direction. Planarity is of obvious benefit in graph drawing, so it is a natural problem to consider upward drawings in combination with planarity.

An *upward plane drawing* of a digraph is a plane drawing such that each directed arc is represented as a curve monotone increasing in the y-direction. In particular, the digraph must be acyclic (a DAG). A digraph is *upward planar* if it has an upward plane drawing. Consider the digraphs in Fig. 1. By convention, the edges in the diagrams in this paper are directed upward unless specifically stated otherwise, and direction

* Received by the editors August 17, 1992; accepted for publication (in revised form) August 29, 1994. This research was supported in part by NSERC and performed while the first author was at the University of Waterloo [15]. A preliminary version of the work was presented at the 2nd ACM–SIAM Symposium on Discrete Algorithms (SODA) [16], and an extended abstract later appeared in [27].

[†] Department of Computer Science, University of Toronto, Toronto, ON M5S 1A4, Canada (mdhutton@cs.utoronto.ca).

[‡] Department of Computer Science, University of Waterloo, Waterloo, ON N2L 3G1, Canada (alubiw@uwaterloo.ca).



Upward planar

Non-upward-planar

FIG. 1. *Upward planar and non-upward-planar digraphs.*

arrows are omitted unless necessary. The digraph on the left is upward planar: an upward plane drawing is given. The digraph on the right is not upward planar—although it is planar, since placing v inside the face f would eliminate crossings, at the cost of producing a downward edge. Kelly [17], Kelly and Rival [18], and Di Battista and Tamassia [7] have shown that for every upward plane drawing, there exists a *straight-line* upward plane drawing with the same faces and outer face, in which every edge is represented as a straight line segment. This is an analogue of the previously mentioned straight-line drawing result for undirected planar graphs. The general problem of recognizing upward planar digraphs has recently been shown to be NP-complete [12]. For the case of single-source single-sink digraphs there is a polynomial-time recognition algorithm provided by Platt's result [19] that such a digraph is upward planar iff the digraph with a source-to-sink edge added is planar. An algorithm to find an upward plane drawing of such a digraph was given by Di Battista and Tamassia [7]. For the special case of bipartite digraphs, upward planarity is equivalent to planarity [6].

In this paper, we will give an efficient algorithm to test upward planarity for single-source digraphs, eliminating the single-sink restriction. For the most part, we will be concerned only with constructing an upward planar *representation*—enough combinatorial information to specify an upward plane drawing without giving actual numerical coordinates for the vertices. This notion will be made precise in §3. We will remark on the extension to a drawing algorithm in §7. Our main result is an $O(n^2)$ algorithm to test whether a given single-source, n-vertex digraph is upward planar, and if so, to give a representation for it which leads to a drawing with known methods. This result is partly based on a graph-theoretic result of Thomassen [24, Them. 5.1].

THEOREM 1.1 (Thomassen). *Let Γ be a plane drawing of a single-source digraph G. Then there exists an upward plane drawing Γ' strongly equivalent to (i.e., having the same faces and outer face as) Γ iff the source α of G is on the outer face of Γ, and for every cycle Σ in Γ, Σ has a vertex β which is not the tail of any directed edge inside or on Σ.*

The necessity of Thomassen's condition is clear: for a digraph G with upward plane drawing Γ', and for any cycle Σ of Γ', the vertex of Σ with highest y-coordinate cannot be the tail of an edge of Σ nor the tail of an edge whose head is inside Σ.

Since a 3-connected graph has a unique planar embedding (up to the choice of the outer face) by Whitney's theorem (cf. [2]), Thomassen concludes that his theorem provides a “good characterization” of 3-connected upward planar digraphs—i.e., puts the class of 3-connected upward planar digraphs in NP intersect co-NP. An efficient algorithm is not given, however (there are potentially an exponential number of possible cycles to check), nor does Thomassen address the issue of non-3-connected digraphs (which could have an exponential number of different planar embeddings).

The problem thus decomposes into two main issues. The first is to describe Thomassen's result algorithmically; we do this in §4 with a linear-time algorithm, which provides an alternative proof of his theorem. The second issue is to isolate the triconnected components of the input digraph and determine how to put the “pieces” back together after the embedding of each is complete. This more complex issue is treated in §6, after a discussion of decomposition properties in §5.

The algorithm for splitting the input into triconnected components and merging the embeddings of each operates in $O(n^2)$ time. Since a triconnected graph is uniquely embeddable in the plane up to the choice of the outer face, and the number of possible external faces of a planar graph is linear by Euler's formula, the overall time to test a given triconnected component is also $O(n^2)$, so the entire algorithm is quadratic.

2. Preliminaries.

In addition to the definitions below, we will use standard terminology and notation of Bondy and Murty [2].

All digraphs in this paper are acyclic unless otherwise stated, and n always denotes the number of vertices in the current digraph. We will use the term *cycle* and the various notions of connectivity with respect to the *underlying undirected graph*, so a digraph G is *connected* if there exists an undirected path between any two vertices in G. For S a set of vertices, $G\backslash S$ denotes G with the vertices in S and all edges incident to vertices in S removed. If S contains a single vertex v, we will use the notation $G\backslash v$ rather than $G\backslash\{v\}$. G is k-connected if it has at least $k+1$ vertices and the removal of at least k vertices is required to *disconnect* the graph. By Menger's theorem [2], G is k-connected iff there exist k vertex-disjoint undirected paths between any two vertices. A set of vertices whose removal disconnects the graph is a *cut-set*. The terms *cut vertex* and *separation pair* apply to cut-sets of size one and two, respectively. A graph which has no cut vertex is *biconnected* (2-connected). A graph with no separation pair is *triconnected* (3-connected). For G with cut vertex v, a *component* of G with respect to v is formed from a connected component H of $G\backslash v$ by adding to H the vertex v and all edges between v and H. For G with separation pair $\{u,v\}$, a *component* of G with respect to $\{u,v\}$ is formed from a connected component H of $G\backslash\{u,v\}$ by adding to H the vertices u and v and all edges between u and v and vertices of H. The edge (u,v), if it exists, forms a component by itself. An algorithm for finding triconnected components[1] in linear time is given in Hopcroft and Tarjan [13] (and also [3]). A related concept is that of *graph/digraph union*: we define $G_1 \cup G_2$ for components with “shared” vertices to be the *inclusive* union of all vertices and edges. That is, for v in both G_1 and G_2, the vertex v in $G_1 \cup G_2$ is adjacent to edges in each of the subgraphs G_1 and G_2.

Contracting an edge $e = (u,v)$ in a graph G results in a graph, denoted G/e, with the edge e removed and vertices u and v *identified*. Inserting new vertices within edges of G generates a *subdivision* of G. A *directed subdivision* of a digraph G results from repeatedly adding a new vertex w to divide an edge (u,v) into (u,w) and (w,v). (Directed) graphs G_1 and G_2 are *homeomorphic* if both are (directed) subdivisions of some other (directed) graph. G is planar iff every subdivision of G is planar [2].

[1] Note that Hopcroft and Tarjan's “components” include an extra (u,v) edge.

In a directed graph, the *in-degree* of a vertex v is the number of edges directed toward v, denoted deg^-v. Analogously, the *out-degree* (deg^+v) of v is the number of edges directed away from v. A vertex of in-degree 0 is a *source* in G, and a vertex of out-degree 0 is a *sink*.

Adopting some poset notation, we will write $u \leq v$ if there is a directed path $u \overset{*}{\rightarrow} v$ of length 0 or more, and $u < v$ ($u \overset{+}{\rightarrow} v$) to emphasize that u and v are distinct. Vertices u and v are *comparable* if $u \leq v$ or $v \leq u$, and *incomparable* otherwise. If (u, v) is an edge of a digraph, then u *dominates* v, u is *incident to* v, and v is *incident from* u.

3. A combinatorial view of upward planarity. As discussed by Edmonds and others (see [11]), a connected graph G is planar iff it has a *planar representation*: a cyclic ordering of edges around each vertex such that the resulting set of *faces* F satisfies $2 = |F| - |E| + |V|$ (Euler's formula). A *face* is a cyclically ordered sequence of edges and vertices $v_0, e_0, v_1, e_1, \ldots, v_{k-1}, e_{k-1}$, where $k \geq 3$, such that for any $i = 0, \ldots, k-1$, the edges e_{i-1} (subscript addition modulo k) and e_i are incident with the vertex v_i and consecutive in the cyclic edge ordering for v_i.

We will say that two plane drawings are *equivalent* if they have the same representation—i.e., the same set of faces. Two plane drawings are *strongly equivalent* if they have the same representation and the same outer face.

One method of combinatorially specifying an upward planar drawing is provided by the following result of (independently) Di Battista and Tamassia [7] and Kelly [17]. They use the concept of a planar *s-t digraph*, defined to be a planar DAG which has a single source s and a single sink t and contains the edge (s, t)—exactly the upward planarity condition of Platt [19] for single-source single-sink digraphs.

THEOREM 3.1 (Di Battista and Tamassia, Kelly). *Let G be a directed acyclic graph. If G is upward planar then edges can be added to it to obtain a planar s-t digraph (i.e., G is a (spanning) subgraph of a planar s-t digraph). Conversely, if edges can be added to G to obtain a planar s-t digraph G', then G is upward planar. Furthermore, for any planar embedding Γ of G' with (s, t) on the outer face, there is an upward plane drawing of G strongly equivalent to Γ with the extra edges removed.*

The final statement was not explicitly given. However, to prove their result, Di Battista and Tamassia give an algorithm which takes a planar *s-t* digraph, finds an *arbitrary* planar representation of it, and outputs an upward plane drawing which respects this embedding, so the statement follows. Their algorithm, which we will require later in the paper, runs in $O(n)$ arithmetic[2] steps ($O(n \log n)$ arithmetic steps for a straight-line drawing).

The disadvantage of this NP characterization in terms of planar *s-t* digraphs is the difficulty of testing it. Thomassen's co-NP condition on single-source digraphs suffers from the same problem. For the case of single-source digraphs, we will give a testable (algorithmic) characterization in the next section.

To provide some motivation for this algorithm, we give another characterization of single-source upward planar digraphs, equivalent to Thomassen's.

First, we define $P(v)$, the *predecessor set of* v, to be the set $\{u : u \leq v \text{ in } G\}$. Notice the set $P(v)$ includes v. Define G_v to be the induced subgraph of G on $P(v)$. For a planar representation Γ of G, define Γ_v to be the planar representation induced by Γ on G_v.

[2] It is important to specify the time in arithmetic steps, because the algorithm is necessarily output sensitive: coordinates can require $\Omega(n)$ bits each [8].

PROPOSITION 3.2. *Given a single source DAG G and a planar representation Γ of G with a specified outer face and source s on the outer face, G has an upward plane drawing strongly equivalent to Γ iff the following condition holds.*

CONDITION 3.3. *For each vertex $v \in V$, v is a sink on the outer face of the planar embedding Γ_v induced by $P(v)$.*

We will often refer to a planar representation Γ satisfying Condition 3.3 as an *upward planar representation* of G.

Since a strongly equivalent upward plane drawing provides the same planar representation Γ and predecessors of v have smaller y-coordinates in the drawing, v must be on the outer face of Γ'_v. Thus the necessity holds. We will complete the proof of this in §5; it is not necessary for the algorithm in the next section.

4. Strongly equivalent upward planarity. Consider the following question: given a single-source acyclic digraph G and a planar representation Γ for G, with s on the outer face of Γ, does G admit an upward planar drawing strongly equivalent to Γ?

Define a *violating cycle* of G with respect to Γ to be a cycle Σ such that every vertex of Σ is the tail of an edge inside or on Σ. This is the condition arising from Thomassen's theorem (1.1). As observed in the introduction, a violating cycle in Γ precludes the existence of an strongly equivalent upward drawing.

We present a linear-time algorithm to test whether G has an upward planar embedding strongly equivalent to Γ with a designated outer face. The algorithm will return the edges necessary to augment G so that sinks occur only on the outer face in the positive case or a violating cycle in the negative case. Since any planar representation of a single-source DAG with the source and all sinks on the outer face is a subgraph of a planar s-t digraph—simply designate one sink as t and add an edge from the source and all other sinks to it—the algorithm provides a new proof of Thomassen's theorem.

The algorithm is recursive, and the proof that it works is by induction. If there is a sink v on the outer face of Γ, then recursively (trivial if G has one node) determine a violating cycle for $G \backslash v$ (in which case we are done) or a set of edges X required to augment $\Gamma \backslash v$ ($G \backslash v$) to a planar representation Γ' with all sinks on the outer face. Now add v and edges incident to v to the outer face of Γ'. To determine the additional required edges to resolve the internal sinks in the new faces, consider all vertices w which are sinks on the outer face of Γ' but are not sinks on the outer face of Γ. Adding the edges (w, v) (where they do not already exist in G) to X retains planarity, single-sourcedness, and acyclicity in $G \cup X$ and does not change the outer face.

It remains to deal with the case when the outer face of Γ has no sink. We claim that in this case G has a violating cycle: If the outer face of Γ is a cycle, then it is a violating cycle. If the outer face is a walk, then follow it starting at s, and let v be the first vertex which repeats. Vertex v must be a cut vertex. Consider the segment of the walk from v to v. If this segment contains only one other vertex, say u, then u is a sink, which is a contradiction. Otherwise, we obtain a cycle C from v to v. The two edges incident with v must be directed away from v, and no other vertex is a sink on C, so C must be a violating cycle.

The above algorithm can be implemented in linear time (so that each vertex is involved in no more than a constant number of operations) using data structures no more complicated than a linked list. We then have Theorem 4.1.

THEOREM 4.1. *Given an n-vertex single-source acyclic digraph G and a plane representation Γ, the above algorithm tests, in linear time, whether G admits an*

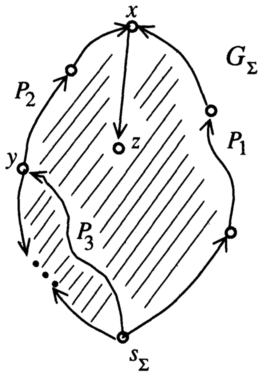


FIG. 2. *Violating cycle precludes Condition* 3.1.

upward planar drawing strongly equivalent to Γ.

5. Decomposition properties of upward planar graphs. This section completes the discussion of upward planar representations and introduces various decomposition properties of upward planar digraphs. The purpose is twofold: first, the properties are necessary for the proofs in the next section; second, they provide an intuitive look at the structure of upward planar digraphs and hence motivate the decomposition approach we take in the recognition algorithm.

We begin by completing the proof of Proposition 3.2 from §3.

Proof of Proposition 3.2: *Sufficiency.* We need that for any planar representation Γ of G satisfying Condition 3.3, G admits an upward plane drawing strongly equivalent to Γ—equivalently, that the existence of a violating cycle precludes Condition 3.3 from holding for some vertex v.

Suppose a violating cycle Σ exists in G with respect to Γ. Let G_Σ be the subgraph of G formed by edges and vertices inside or on Σ.

Without loss of generality, G_Σ has one source s_Σ, which must lie on Σ. If G_Σ had two sources s_1 and s_2, then both would be on Σ. Since G has a single source s, there exist directed paths P_1 from s to s_1 and P_2 from s to s_2. The last edge of each path is not in G_Σ. If either path has a vertex other than its terminal vertex on Σ, then adding to Σ the portion of the path from the last such vertex to the terminal produces another violating cycle enclosing a larger subgraph with one fewer sources. Otherwise, P_1 and P_2 contain a last common vertex, and adding the portions of the paths from that vertex to the terminals s_1 and s_2 produces another violating cycle enclosing a larger subgraph with one fewer sources. Thus we can assume that G_Σ has a single source s_Σ. The remainder of the proof references Fig. 2.

Starting from s_Σ, walk counterclockwise around Σ. Let x be the first encountered vertex with both edges of Σ directed toward x. Let y be the first encountered vertex after x with both edges of Σ directed away from y. Note that both exist, although y may be s_Σ. Let P_1 be the directed path from s_Σ to x, counterclockwise on Σ, and let P_2 be the directed path from y to x clockwise on Σ.

Since s_Σ is the single source of G_Σ, there is a directed path P_3 in G_Σ from s_Σ to y. (If $y = s_\Sigma$, then P_3 is this single vertex.) P_3 cannot contain a vertex of P_2 other than y; otherwise we would get a directed cycle using portions of P_2 and P_3. Let u be the last vertex of P_1 on P_3. Let Π be the simple undirected cycle consisting of the portion of P_3 from u to y, the portion of P_2 from y to x, and the portion of P_1 from

u to x. Let G_Π be the subgraph of G formed by edges and vertices inside or on Π.

Since Σ is a violating cycle, x is not a sink in G_Σ, so there is an edge (x, z) inside Σ and thus inside Π. We will show that vertex z violates condition 3.1. Vertex z cannot be on Π; otherwise, a directed cycle is formed. Thus z is strictly inside Π. But all the vertices of Π are predecessors of z. Thus z violates condition 3.1. □

Note that the results of the preceding two sections, combined with the characterization of Di Battista and Tamassia and the single-source characterization of Thomassen, give the following theorem.

THEOREM 5.1. *The following conditions are equivalent for a single-source DAG G with planar representation Γ having a designated outer face and single source s which is on the outer face:*

(i) *G has an upward plane drawing strongly equivalent to Γ.*

(ii) *G is a (spanning) subgraph of some planar s-t digraph which has an upward plane drawing strongly equivalent to Γ (after removal of the extra edges).*

(iii) *For all $v \in G$, v is a sink on the outer face of Γ_v.*

(iv) *Γ does not contain a violating cycle.*

We note that condition (iii) is the only one which can obviously be tested in polynomial time.

In the remainder of this section, we give some operations which preserve upward planarity. The first operation contracts an edge connected to a vertex of in- (out-) degree 1. The second attaches one upward planar digraph to another at a single vertex. The third attaches an upward planar digraph in place of an edge of another upward planar digraph. The last splits a vertex into two vertices.

First, we will prove a useful preliminary result.

PROPOSITION 5.2. *Let G be a connected upward planar digraph. Then G is a subgraph of some single-source upward planar G^* such that all nonsource $v \in V(G)$ have the same in-degree in G as in G^*.*

Proof. We illustrate how to add the edges required to "resolve the extra sources" without affecting the in-degree of nonsource vertices.

Let Γ be a drawing of G in the plane bounded by $x_{\min}$, $x_{\max}$, $y_{\min}$ and $y_{\max}$, with height h and width w and centred at $(0,0)$. Without loss of generality, we assume Γ is a straight-line drawing. Add new vertices s, t, l, and r at $(0,-2h), (0,2h), (-2w,0)$, and $(2w,0)$, respectively. Add lines (edges) $(s,l), (s,r), (r,t)$, and (l,t). Add further edges (s,w) for all vertices w drawn with y-coordinate of $y_{\min}$ and (w,t) for all vertices w drawn with y-coordinate of $y_{\max}$.

The construction so far has merely added a specified outer face on the drawing, with a unique maximum sink and minimum source, so clearly the resulting drawing Γ^{**} (digraph G^{**}) is an upward plane drawing (upward planar digraph). We now wish, for each source x, to "resolve" the source by adding a new edge incident to it; the resulting digraph will prove our proposition. Let Γ be an upward plane drawing of G, and perform the following operation for each source x, except the one just added in the outer face. Extend a line L vertically down from x to the first line or vertex in the drawing. If L first intersects a vertex w, add the (w,x) edge to both G and the drawing Γ—the result is clearly upward planar. If L first intersects an edge, rotate it along the edge until L hits some vertex w of Γ and add the edge (w,x) as before; for one of the two directions, a vertex will be found before the line becomes horizontal. Neither operation added in-degree to a nonsource vertex, so the claim is satisfied. □

Note that the above is of no particular *algorithmic* significance, since the drawing

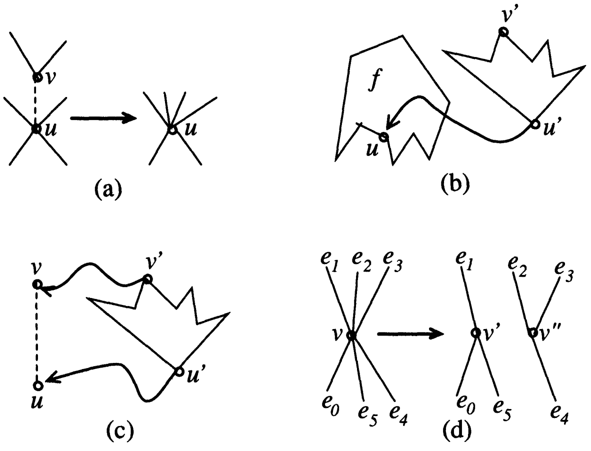


FIG. 3. *Properties of upward planar representation.*

of G (or existence thereof) is the goal rather than the input. However, it allows us to prove the following lemmas in the more general context of upward planar digraphs, i.e., without the single-source assumption.

LEMMA 5.3. *Let G be a DAG and v, dominated by u, be a vertex of G with in-degree* 1. *Then $G/(u,v)$ is upward planar if G is upward planar.* (*See Fig.* 3(a).)

Note that the same result holds for G and edge (u,v) with $deg^+u = 1$ by symmetry. Lemma 5.3 is a generalization of a previously known fact—that G is upward planar iff any directed subdivision of G is (cf. [24]).

Proof. Let Γ be an upward plane drawing of G. Applying Proposition 5.2, there is a single-source digraph G^* containing G as a subgraph in which the in-degree of v is still 1. Clearly if the result holds for $G^*/(u,v)$, it holds for any subgraph, namely $G/(u,v)$, so it will be sufficient to assume G is a single-source digraph for the remainder of the proof and show Condition 3.3 holds.

Let Γ be a planar representation for G, with a designated outer face, satisfying Condition 3.3. Let Γ_w for $w \in V$ be as defined for Condition 3.3. Then Γ', formed by contracting (u,v) in Γ, is a planar representation for $G' = G/(u,v)$ with a designated outer face. Clearly if some $w \neq v$ is on the outer face of Γ_w, it is on the other face of $\Gamma_w/(u,v)$. This, with the fact that G is acyclic, implying $G_w/(u,v) = G'_w$ for all $w \in V - \{v\}$ (i.e., P(w) doesn't change as a result of contracting (u,v)), gives that w is a sink on the outer face of Γ'_w—Condition 3.3. □

Topologically, this construction can be viewed as "pulling" v and all edges incident from v down a corridor of width ϵ around (u,v) until u and v meet.

LEMMA 5.4. *Let G be an upward planar digraph with a vertex u, and let H be an upward planar digraph with a single source u'. Let G' be the digraph formed by identifying u and u' in $G \cup H$. Then G' is upward planar.* (*See Fig.* 3(b).)

Proof. As above, there is an upward planar single-source digraph G^* containing G by Proposition 5.2, and G' is a subgraph of $G^* \cup H$ with u and u' identified, so it is sufficient to prove the result for a single-source upward planar G.

Suppose Γ^G and Γ^H are the given planar representations, with designated outer faces, both satisfying Condition 3.3. Let Γ^G_v and Γ^H_v be as defined for Condition 3.3 for G and H, respectively. We show how to construct a planar representation Γ' for $G \cup H$ (identifying u and u'), with a given outer face, which satisfies Condition 3.3.

If u is on the outer face of Γ^G, then place Γ^H in the outer face, identifying u and u'. Otherwise, there are (possibly) $k+1$ faces of Γ^G corresponding to the outer face of Γ^G_u (for k the out-degree of u); insert Γ^H in any one of these faces. It is easy to show that, under this construction, w is a sink on the outer face of Γ'_w if it was a sink on the outer face of Γ^G_w (respectively, Γ^H_w) previously. □

This construction can be viewed as "inserting" the drawing for H into some face "above" u in the drawing of G.

LEMMA 5.5. *Let G be an upward planar digraph with an edge (u,v), and let H be an upward planar digraph with a single source u' and a sink v' both on the outer face. Let G' be the digraph formed by removing the (u,v) edge of G and adding H, identifying vertex u with u' and vertex v with v'. Then G' is upward planar.*

Proof. This has the same flavour as the previous proof, so we can be more brief. Again, by Proposition 5.2 it is sufficient to assume that G has a single source. Let Γ^G and Γ^H be planar representations, with designated outer faces, satisfying Condition 3.3.

Form a planar representation Γ' of G' by replacing (u,v) by Γ^H in Γ^G. The result is planar and has a well-defined outer face. We need that Γ' satisfies Condition 3.3. As in the previous proof, it is easy to show that w is a sink on the outer face of Γ'_w whenever it is a sink on the outer face of Γ^G_w (respectively, Γ^H_w). □

This construction can be viewed as replacing a directed edge in an upward plane drawing of G with another upward plane drawing of H which is, in some sense, "topologically equivalent" to an edge within the drawing of G.

LEMMA 5.6. *Let G be a DAG which has an upward planar representation where the cyclic edge order about vertex v is $e_0,\ldots,e_{k-1}$ (vertices $v_0,\ldots v_{k-1}$). Let G' be the DAG formed by splitting v into two vertices: v' incident with edges $e_i,\ldots,e_j$, and v'' incident with edges $e_{j+1},\ldots,e_{i-1}$ ($i \neq j$, arithmetic* mod *k). Then G' is upward planar. If G has a single source, and i and j are such that each of v' and v'' retains at least one incoming edge, then the resulting G' is also a single-source digraph.* (*See Fig.* 3(d).)

Proof. The last statement is clearly true; no new sources can be added by the construction if each new vertex has an incoming edge. Again, it is sufficient to show the first part for single-source G, since the resulting digraph is otherwise a subgraph of the construction applied to G^* (of Proposition 5.2). Let Γ be a planar representation for G satisfying Condition 3.3.

Without loss of generality, assume that the construction does not make v' a source unless v was itself a source. We prove the result for $G'' = G' + (v', v'')$—it is easy to augment Γ' (the planar representation formed by separating v into v' and v'' in Γ) to Γ'' with the edge (v', v'') because v' and v'' share a face. The construction of Γ'' from Γ preserves planarity and cannot introduce a dicycle; it remains to show Condition 3.3 holds for G'' and Γ''. The set of faces in Γ'' is identical to that of Γ, save for the two new faces sharing (v', v''), so Condition 3.3 is satisfied for all w not incident from either v' or v''. Any v_i incident from v' or v'' (via edge e_i) is clearly on the outer face of Γ''_{v_i} whenever it is on the outer face of Γ_{v_i}, since the construction can only add vertices to an outer face, never remove them. □

6. Separation into triconnected components. The algorithm of §4 tests for upward planarity of a single-source DAG G starting from a given planar representation and an outer face of G. In principle, we could apply this test to all planar representations of G, but this would take exponential time. In order to avoid this, we will decompose the digraph into biconnected and then into triconnected components.

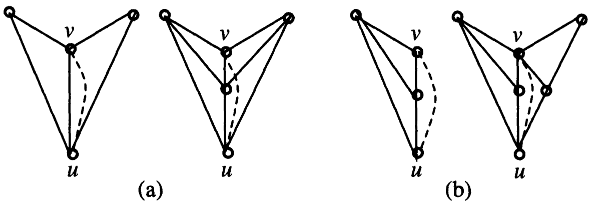


FIG. 4. *Added complication of two-vertex cut-sets.*

Each triconnected component has a unique planar representation (see [2]) and only a linear number of possible outer faces. We can thus test upward planarity of the triconnected components in quadratic time using the algorithm of §4. Since we will perform the splitting and merging of triconnected components in quadratic time, the total time will then be quadratic.

To decompose G into biconnected components we use the following lemma.

LEMMA 6.1. *A DAG G with a single source s and a cut vertex v is upward planar iff each of the k components H_i of G (with respect to v) is upward planar.*

Proof. If G is upward planar, then so are its subgraphs, the H_i's. For the converse, note that if $v \neq s$, then v is the unique source in all but one of the H_i's; and if $v = s$, then v is the unique source in each H_i. Apply Lemma 5.4. □

Dividing G into triconnected components is more complicated, because the cut-set vertices impose restrictive structure on the merged digraph. In the biconnected case, it is sufficient to simply test each component separately, since biconnected components do not interact in the combined drawing. The analogous approach for triconnected components would be to add a new edge between the vertices of the cut-set in each component, then perform the test recursively. This, however, does not suffice for upward planarity, as illustrated by the two examples in Fig. 4. (Recall our convention that direction arrowheads are assumed to be "upward" unless otherwise specified.) In (a), the union of the digraphs is upward planar, but adding the edge (u, v) to each makes the second component non-upward-planar. In (b), the digraph is non-upward-planar, but each of the components is upward planar with (u, v) added.

We will find it convenient to split the digraph G into exactly two pieces at a separation pair $\{u, v\}$, where one of these pieces, E, is a component with respect to the separation pair, and the other piece, F, is the union of the remaining components. This forces each piece to fit into one face of the embedding of the other piece.

LEMMA 6.2. *For G, E, and F as above, let Γ be a plane embedding of G, and let Γ_E and Γ_F be the embeddings induced on E and F, respectively. Then in Γ, all of E lies in a single face of Γ_F and all of F lies in a single face of Γ_E. Furthermore, at least one of E, F must lie in the outer face of Γ_F, Γ_E, respectively.*

Proof. Any distinct vertices x and y in E, neither being u or v, must share a path in E which avoids both u and v (lies entirely within E). Hence, for Γ, a plane embedding of G, and Γ_E and Γ_F, the respective subembeddings of E and F, if vertices x and y of E are in different faces of Γ_F, they could not share a path which avoids both u and v without violating planarity. Clearly, also, one of E, F must have two vertices on the outer face of the total drawing Γ (which has at least three vertices) and hence must lie entirely in the outer face of the other subdrawing. □

We will test the upward planarity of a biconnected digraph G by breaking it at a cut-set into pieces E and F, as above, and looking for upward planar embeddings Γ_E and Γ_F that fit together as in Lemma 6.2.

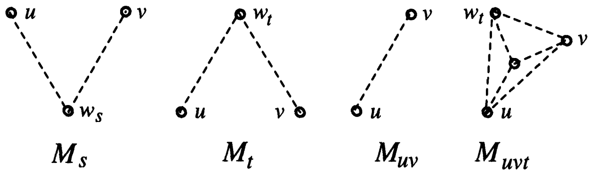


FIG. 5. *Marker graphs.*

We need a face in Γ_E that contains u and v and is the right "shape" to accommodate the "shape" of Γ_F; and we need a face in Γ_F that contains u and v and is the right "shape" to accommodate the "shape" of Γ_E. (Figure 4(b) showed an example where these conditions fail.) These conditions will be enforced by adding a "marker" connecting u and v to E (F, respectively) that captures the "shape" of Γ_F (Γ_E, respectively) and forces u and v to lie in a common face. For example, the simplest case is when u is the source and v is the sink of F; then the marker representing Γ_F in E is a single (u, v) edge.

Besides playing the primary role described above, the markers will also be used to make the two components 3-connected and single source, thus allowing us to recurse on smaller subproblems. The markers we are interested in are shown in Fig. 5.

We need one other main idea. The last statement of Lemma 6.2 is that one of E, F, must lie in the outer face of Γ_F, Γ_E, respectively. For undirected digraphs, this causes no problem, since any face can be made the outer one. However, for upward planarity, this condition complicates things. The situation is simplified when $s \neq u, v$. In this case, we will take E to be the $\{u, v\}$ component containing s, and so Γ_E must lie in the outer face of Γ_F. When $s \in \{u, v\}$, we must do extra work to decide the "outer" component.

Having determined or decided that Γ_E must lie in the outer face of Γ_F, we know that u and v must be on the outer face of Γ_F. Thus our algorithm will solve the more general problem of testing upward planarity under the condition that some specified set X of vertices, called the "outer" set, must lie on the outer face.

To summarize, given a biconnected digraph G and an "outer" set of vertices X, we break G at a cut-set $\{u, v\}$ into one component E, containing s, and the union of the remaining components F. We add appropriate markers to E and F, specify their "outer" sets, and recurse. We must prove that G has an upward planar embedding with its "outer" set on the outer face iff the smaller digraphs do.

The details and proofs of this plan make up the remainder of this section. We will consider three cases separately: when u and v are incomparable, when u and v are comparable with $s < u < v$, and when u and v are comparable with $u = s$.

An important note to make at this time is that the markers, except for M_{uv}, are subgraphs attached at only two vertices, which means that $\{u, v\}$ will still constitute a cut-set. For the purposes of determining cut-sets and making recursive calls, the markers should be treated as distinguished edges—a single edge labelled to indicate its role. As long as the type of marker is identified, the algorithm can continue to treat the vertices of attachment as source, sink, or neither, as appropriate for the particular operation.

6.1. Cut-set u, v; u and v are incomparable. Here we consider vertex cut-sets $\{u, v\}$ which are incomparable (then neither is s). We divide the digraph G at $\{u, v\}$ into two subgraphs—the *source component* E (the one component which contains the source s) and the union of the remaining components F.

First, we need some preliminary results.

PROPOSITION 6.3. *If G is a connected DAG with exactly two sources u and v, then there exists some w_t such that two vertex-disjoint (except at w_t) directed paths $u \overset{+}{\rightarrow} w_t$ and $v \overset{+}{\rightarrow} w_t$ exist in G.*

Proof. Let G be such a DAG and let P be an undirected path from u to v. Note that every x in P is comparable with either u or v, otherwise G has more than two sources. Follow P from u to the first node x (following y on P) incomparable with u (in G). Then x is comparable with v and (x, y) is an edge in G (otherwise $u < x$), so y is also comparable with v. Taking the first common vertex in the paths $u \overset{+}{\rightarrow} y$ and $v \overset{+}{\rightarrow} y$ gives w_t. □

The following result shows the existence of lower bounds and upper bounds (in the partial order corresponding to G) under certain conditions. This allows us to prove the necessity conditions in Theorem 6.5 (to come).

LEMMA 6.4. *If G is a biconnected DAG with a single source s, and if u and v are incomparable vertices in G, then there exists some w_s such that two vertex-disjoint (except at w_s) directed paths $w_s \overset{+}{\rightarrow} u$ and $w_s \overset{+}{\rightarrow} v$ exist in G. If $\{u, v\}$ is a cut-set in G, then there also exists some w_t such that two vertex-disjoint (except at w_t) directed paths $u \overset{+}{\rightarrow} w_t$ and $v \overset{+}{\rightarrow} w_t$ exist in G.*

Proof. Since G is a single-source digraph, there exist directed paths from s to u and s to v in G. Taking the last common vertex in these paths gives w_s.

For the existence of w_t, let u and v be an incomparable separation pair of G. Since $\{u, v\}$ cuts G into at least two connected components, any nonsource component H has u and v as its (exactly) two sources, and the result follows from Proposition 6.3. □

We are now ready to proceed with the statement of the first main result of the decomposition.

THEOREM 6.5. *Let G be a biconnected directed acyclic digraph with a single source s, and let $X = \{x_i\} \subseteq V(G)$ be a set of vertices. Let $\{u, v\}$ be a separation pair of G, with u and v incomparable. Let E be the connected component of G with respect to $\{u, v\}$ containing s, and let F be the union of all other components. Then G admits an upward plane drawing with all vertices of X on the outer face iff*

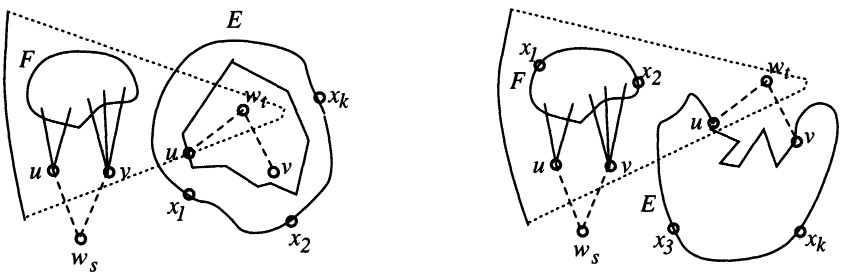
(i) *$E' = E \cup M_t$ admits an upward plane drawing with all vertices of X in E on the outer face and with w_t on the outer face if some $x \in X$ is contained in F, and*

(ii) *$F' = F \cup M_s$ admits an upward plane drawing with all vertices of X in F on the outer face.*

Here, as in the remaining cases, the proof will have the same basic flavour. The necessity of the marker conditions will follow from the existence of the corresponding marker "within" (i.e., homeomorphic to a subgraph of) the companion component. The sufficiency will be shown by applying the properties of an upward planar representation from §4 to combine upward planar representations for the two subproblems into a single upward planar representation.

Proof. Necessity. Suppose G admits an upward planar drawing (representation) Γ with all $x_i \in X$ on the outer face. Follow Fig. 6.

Since u and v are incomparable, there exists a w_s and vertex-disjoint directed paths $w_s \overset{+}{\rightarrow} u$ and $w_s \overset{+}{\rightarrow} v$ in G by Lemma 6.4; specifically, these must be in E if G has a single source. Then $F' = F \cup \{(w_s, u), (w_s, v)\}$ is homeomorphic to a subgraph



(a) No x_i's in F

(b) x_i's in F

FIG. 6. *Merging E and F; cut-set $\{u, v\}$ is incomparable.*

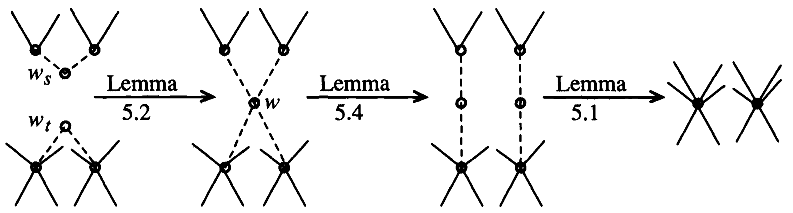


FIG. 7. *Merge construction; $\{u, v\}$ is incomparable.*

of G and hence upward planar itself. F' can be obtained from G by deleting and contracting E to its marker—this will not decrease the set of vertices on the outer face. Thus, since Γ has all the x_i's of F on its outer face, F' has an upward planar drawing with all vertices of X in F on the outer face. We have (ii).

Similarly, there exists some w_t in F such that $E \cup \{(u, w_t), (v, w_t)\}$ is homeomorphic to a subgraph of G, so E' is upward planar. As above, any x_i in E and also on the outer face of Γ will be on the outer face of the subdrawing formed by Γ_E and the $u \overset{+}{\rightarrow} w_t, v \overset{+}{\rightarrow} w_t$ paths. By Lemma 6.2, Γ_F must lie entirely within one face of Γ_E, so if some x_i in F is on the outer face of the drawing Γ, then the portion of the drawing formed by the $u \overset{+}{\rightarrow} w_t$ and $v \overset{+}{\rightarrow} w_t$ paths (hence w_t itself) must be in the outer face of the subdrawing Γ_E, giving (i).

Sufficiency. Suppose E' and F' admit upward planar representations satisfying (i) and (ii). Identifying the single source w_s of F' and w_t in E' (call the new vertex w) per Lemma 5.4, the result G' is upward planar. Splitting w into w_l with the leftmost two vertices and w_r with the rightmost two vertices (Lemma 5.6) and contracting the (w_l, u) and (w_l, v) edges (Lemma 5.3) gives exactly G, which is hence upward planar. The construction is illustrated in Fig. 7.

For the sufficiency of the x_i conditions, we notice that all nonmarker vertices of X on the outer face of the E' drawing are also on the outer face of the constructed drawing. If F contains no x_i, this is sufficient; otherwise, w_t being on the outer face of E' guarantees that the nonmarker vertices of X in F' are also on the outer face of the result. □

6.2. Cut-set u, v, where $u < v$, $u \neq s$. Here we consider any other vertex cut-sets not involving the source s. We again divide G into the *source component* E and the union of the remaining components F. Note that v can be a source in E as long as there is a u to v path in F.

An additional preliminary result will be useful.

LEMMA 6.6. *If G is a biconnected DAG with a single source s and cut-set $\{u,v\}$, where $u < v$ in G and $u \neq s$, then in any nonsource component H of G with respect to $\{u,v\}$, where $deg^+v > 0$, there exists some w_t such that $u \xrightarrow{+} w_t$ and $v \xrightarrow{+} w_t$ are vertex-disjoint directed paths in H.*

Proof. No vertex other than u and v can be a source in H, otherwise G has more than one source; and u is always a source in H. If v is also a source, then we are done by Proposition 6.3.

If v is not a source, let $w \in H$ be a vertex dominated by v. G is biconnected, so there are two vertex-disjoint $u \xrightarrow{+} w$ undirected paths in G. But u and v are cut vertices in G, so at least one of the paths P lies completely within H and does not contain v (since w is in H and the only exit points from H are u and v). Every x on P is comparable with either u or v, or else G has more than one source. Find the last vertex y on P which has a $u \xrightarrow{+} y$ path (in G) without v. If $y = w$, then we are done. Otherwise, the vertex x following y on P has any $u \xrightarrow{+} x$ path necessarily going through v. Then there exist directed paths $v \xrightarrow{+} x$ and $u \xrightarrow{+} x$ with the latter not containing v, so the first common vertex on these paths provides a w_t. □

We can now continue with the second main result of the decomposition.

THEOREM 6.7. *Let G be a biconnected directed acyclic digraph with a single source s, and let $X = \{x_i\} \subseteq V(G)$ be a set of vertices. Let $\{u,v\}$ be a separation pair of G with $u < v$ in G and $u \neq s$. Let E be the source component of G with respect to $\{u,v\}$ and F be the union of all other components. Then G admits an upward plane drawing with all vertices of X on the outer face iff*

(i) *$E' = (E \cup F$-marker) admits an upward plane drawing with all vertices of X in E on the outer face and w_t (if it exists; otherwise the edge (u,v)) on the outer face if some $x \in X$ is contained in F, and*

(ii) *$F' = (F \cup E$-marker) admits an upward plane drawing with w_t (if it exists; otherwise the edge (u,v)) and all vertices of X in F on the outer face,*

where

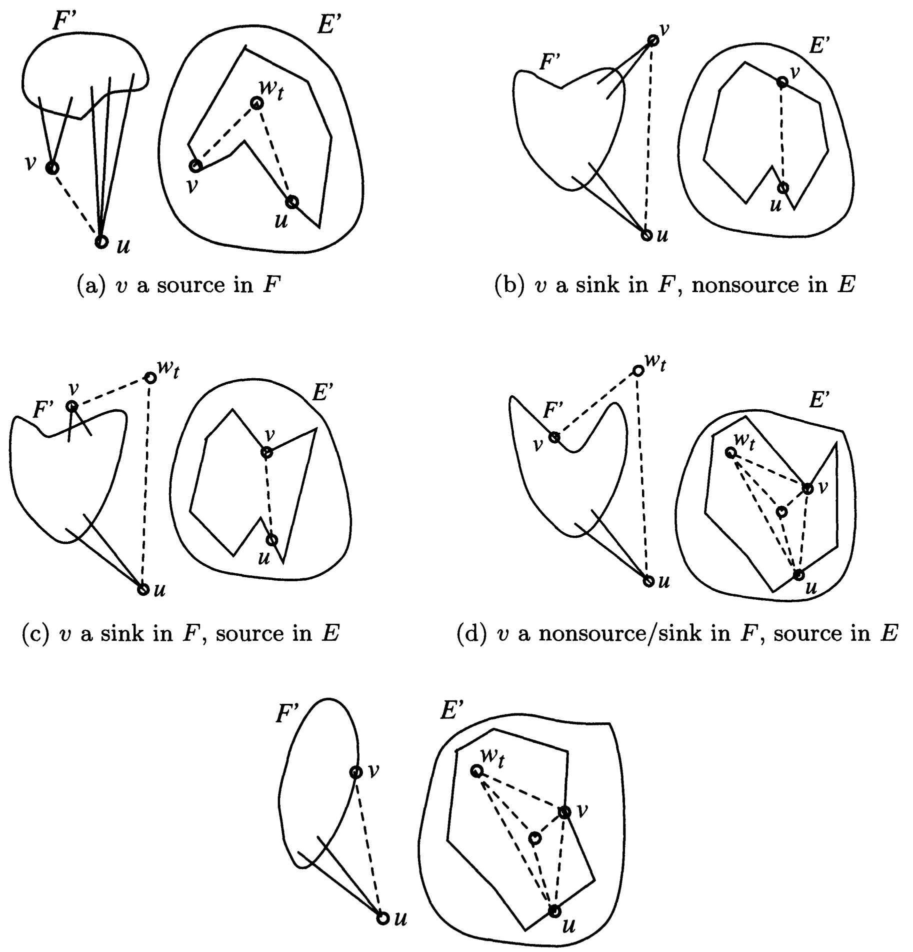
$$F\text{-}marker = \begin{cases} M_t & \text{if } v \text{ is a source in } F, \\ M_{uv} & \text{if } v \text{ is a sink in } F, \\ M_{uvt} & \text{otherwise,} \end{cases}$$

and

$$E\text{-}marker = \begin{cases} M_t & \text{if } v \text{ is a source in } E, \\ M_{uv} & \text{otherwise.} \end{cases}$$

Proof. Necessity. Suppose G admits an upward plane drawing with all $x_i \in X$ on the outer face. Follow Fig. 8.

Necessity of condition (i). If v is a source in F, then there exists some w_t in F and vertex-disjoint paths $u \xrightarrow{+} w_t$ and $v \xrightarrow{+} w_t$ by Proposition 6.3; so $E' = E \cup M_t$ is homeomorphic to a subgraph of G and is upward planar. If v is a sink in F, then u is the single source of F, since only u and v are possible sources. Thus, in F, there is a path $u \xrightarrow{+} v$, so $E' = E \cup M_{uv}$ is homeomorphic to a subgraph of G and is upward planar. If v is neither a source nor a sink in F, then, by Lemma 6.6, there is also some $w_t > v$ and disjoint directed paths $u \xrightarrow{+} w_t$ and $v \xrightarrow{+} w_t$ in G. Since v is a nonsource in F, there is also a $u \xrightarrow{+} v$ path in F. This path crosses the $u \xrightarrow{+} w_t$ path at some latest



(a) v a source in F

(b) v a sink in F, nonsource in E

(c) v a sink in F, source in E

(d) v a nonsource/sink in F, source in E

(e) v a nonsource/sink in F, nonsource in E

FIG. 8. *Merging E and F; cut-set $\{u, v\}$ and $u < v$.*

vertex z on that path, so $E \cup (u \xrightarrow{*} z) \cup (z \xrightarrow{+} v) \cup (z \xrightarrow{+} w_t) \cup (v \xrightarrow{+} w_t)$ is a subgraph of G and hence upward planar. Note that these four paths are disjoint. Since z has in-degree 1, we can contract the $u \xrightarrow{*} z$ path to u without destroying upward planarity, by Lemma 5.3, so $E \cup \{(u, v), (u, w_t), (v, w_t)\}$ has an upward planar subdivision and is upward planar itself. By Lemma 6.2, F lies in a single face of Γ_E, so no other vertices lie inside the u, v, w_t triangle, and the extra edges and vertex for M_{uvt} can be added without destroying planarity.[3] If some x_i is in F, then by the same argument as Theorem 6.5, all of F must be in the outer face of Γ_E. The marker, hence w_t or the (u, v) edge as appropriate, is therefore in the outer face of the drawing induced by E on Γ.

Necessity of condition (ii). If v is a source in E, then by Proposition 6.3, there are vertex-disjoint paths $s \xrightarrow{+} w_t$ and $v \xrightarrow{+} w_t$ in E. There must be an $s \xrightarrow{+} u$ path in E, otherwise there is either a second source (u is a source in F, so it cannot

[3] The point of adding these edges is to fix the face in E for the sufficiency conditions.

also be a source in E) or a cycle in G ($u < v$ in G, so there can be no $v \overset{+}{\rightarrow} u$ directed path in E). Let z be the last vertex common to paths $s \overset{+}{\rightarrow} u$ and $s \overset{+}{\rightarrow} w_t$. Then $F \cup \{(z, u), (z, w_t), (v, w_t)\}$ is homeomorphic to a subgraph of G and is upward planar. Since $deg^- u = 1$ (in this digraph), the edge (z, u) can be contracted without destroying upward planarity, by Lemma 5.3, and $F' = F \cup M_t$ is upward planar.

Otherwise (v a nonsource), if $u < v$ in E, then $F' = F \cup M_{uv}$ is homeomorphic to a subgraph of G and hence is upward planar. If u and v are incomparable in E, then they share a greatest lower bound w_s, by Lemma 6.4, and $F \cup \{(w_s, u), (w_s, v)\}$ is upward planar. Again, $deg^- u = 1$ in F, so the (w_s, u) edge can be contracted to give $F' = F \cup M_{uv}$.

The requirement for the E-marker to be on the outer face of Γ induced by F follows as before. Γ_E lies entirely within one face of Γ_F, and this is necessarily the outer face since E contains the source s.

Sufficiency. Suppose E' and F' admit upward plane drawings meeting the requirements (i) and (ii).

Case 1: *v is a source in F.* (See Fig. 8(a).) If v is a source in F, it cannot at the same time be a source in E, since $u < v$ in either E or F. Thus $F' = F \cup (u, v)$ is upward planar with single source u. Using Lemma 5.4, add F' (with u and v renamed as u' and v') to E', identifying u' with w_t. We can do this so that edges (v, w_t) and (w_t, v') are consecutive in the cyclic order about w_t. Using Lemma 5.6, split w_t by making these two edges incident with a new vertex u_1 and the remaining edges incident with a new vertex u_2. Now v' and u_1 have in-degree 1, so use Lemma 5.3 to contract their in-edges, thus identifying v and v'. Vertex u_2 has in-degree 1, so contract (u, u_2). The result is the digraph G, and thus G is upward planar.

Case 2: *v is a sink in F.* (The two possibilities are illustrated in Fig. 8(b) and (c).) If v is a nonsource in E, then $F' = F \cup (u, v)$ is upward planar with u and v on the outer face by assumption. If v is a source in E, then $F' = F \cup M_t$ is upward planar with w_t on the outer face. In either case, F is upward planar with single source u and sink v on the outer face. By Lemma 5.5, we can add F to E' in place of the (u, v) edge in E', and the result, G, is upward planar.

Case 3: *v is a nonsource/sink in F.* (See Fig. 8(d).) Suppose v is a source in E. Then $F' = F \cup M_t$ is upward planar with the sink w_t on the outer face. Using Lemma 5.5, add F' (renaming u and v to u' and v' respectively) to E' in place of the edge (u, v), identifying u' with u and w_t with v. Throw away the edge (u, w_t) and the remaining marker edges of E'. Vertex v now has in-degree 1, so the edge (v', v) can be contracted by Lemma 5.3, and the result, G, is upward planar. Note that the M_{uvt} marker attached to E' is stronger than we actually require here (M_{uv} would do), but it necessarily does exist (as previously proven) and is needed for the next part of this case.

Suppose then that v is a nonsource in E. Consider the upward planar representation of E' and throw away the marker edges, save for $(u, w_t), (v, w_t)$, and (u, v), which then form a face. $F' = F \cup (u, v)$ is upward planar with u and v on the outer face. Let z be some sink on the outer face, and add the edge (v, z) to obtain F'', upward planar with u, v, and z on the outer face. Using Lemma 5.5, add F'' (with u and v renamed to u' and v') to E' in place of the edge (u, w_t), identifying u' with u and z with w_t. Do this so that v' and v share the face of edges $(u, v'), (v', z), (u, v)$, and (v, z). Clearly, we can now identify the vertices v and v'. We obtain an upward planar digraph containing G as a subgraph. See Fig. 8(e).

As in the proof of Theorem 6.5, we notice that all vertices on the outer face of E' are necessarily on the outer face of the combined drawing, and if some x_i exists in F, then it is on the outer face of F' and is forced to the outer face of the combined drawing by the second part of condition (i). □

6.3. Cut-set *s*, *v*. As mentioned in the introduction to §6 (see also Lemma 6.2), it is important to be able to distinguish the "inner" and "outer" components. The inner component will be embedded in a face of the outer one, and thus the inner component will have to have the marker on its outer face since this marker is a proxy for the outer component. If we have to check each component as a potential inner component, we must recursively solve two subproblems for each component, and an exponential-time blowup results.

Until now, the outer component has been uniquely identified as the *source component*, since that component cannot lie within an internal face of any other component. If we have a cut-set of the form $\{s, v\}$, where s is the source, then we lose this restriction, so we handle it instead by requiring one of the components, E, to be 3-connected, so that deciding if it can be the inner face does not require recursive calls. To decide if E can be the inner face, we need to test if it satisfies the role of E in the previous theorem—i.e., has an upward planar representation with the marker on its outer face. This can be done in linear time using the algorithm of §4. If G has only cut-sets of the form $\{s, v\}$, then for at least one such cut-set, one of the components will be triconnected. Given the list of cut-sets, we can find such a cut-set and such a component in linear time using a depth-first search.

We capture these ideas in terms of two theorems. One is applicable if the triconnected component, E, can be the inner component, and one if it cannot. E "can be" the inner component iff it satisfies the same conditions that the inner component F satisfied in the previous Theorem 6.7. The similarity of both of these theorems to Theorem 6.7 should be clear. Note that in the statement of these theorems, we continue to use u (redundant since $u = s$) for consistency with previous usage.

THEOREM 6.8. *Let G be a biconnected DAG with a single source s, and let $X = \{x_i\} \subseteq V(G)$ be a set of vertices. Let $\{u, v\}$ be a separation pair of G where $u = s$, E be a 3-connected component of G with respect to $\{u, v\}$, and F be the union of all other components of G with respect to $\{u, v\}$. If*

$E' = (E \cup F$-marker$)$ admits an upward plane drawing with w_t (if it exists; otherwise the edge (u, v)) and all vertices of X in E on the outer face,

then G admits an upward plane drawing with all vertices of X on the outer face iff

(i) *$F' = (F \cup E$-marker$)$ admits an upward plane drawing with all vertices of X in F on the outer face, and w_t (if it exists; otherwise the edge (u, v)) also on the outer face if some $x \in X$ contained in E,*

where

$$E\text{-marker} = \begin{cases} M_t & \text{if } v \text{ is a source in } E, \\ M_{uv} & \text{if } v \text{ is a sink in } E, \\ M_{uvt} & \text{otherwise,} \end{cases}$$

and

$$F\text{-marker} = \begin{cases} M_t & \text{if } v \text{ is a source in } F, \\ M_{uv} & \text{otherwise.} \end{cases}$$

Proof. The proof is similar to the necessity of Theorem 6.7 with some (simplifying) modifications. We will be more brief except where differences exist.

Necessity. Suppose G has an upward planar representation Γ with all x_i's on the outer face, and that some upward planar representation $\Gamma_{E'}$ satisfies the precondition for E'. We need to show that the condition (i) holds.

If v is a source in E, then there exist vertex disjoint paths $u \overset{+}{\to} w_t$ and $v \overset{+}{\to} w_t$ in E by Proposition 6.3, so $F' = F \cup M_t$ is homeomorphic to a subgraph of G and hence upward planar. If v is a sink in E, then u is the only source, so $u \overset{+}{\to} v$ exists in E and $F' = F \cup M_{uv}$ is upward planar. Otherwise, v is neither and there exist vertex-disjoint $u \overset{+}{\to} v$, $v \overset{+}{\to} w_t$, and $u \overset{+}{\to} w_t$ paths in E by Lemma 6.6 and the fact that $u = s$ is the single source in E, so $F' = F \cup \{(u,v),(u,w_t),(v,w_t)\}$ is upward planar. Since these three edges are a face in the representation of this digraph, the required extra edges and vertices can be added and the result is upward planar. The necessity of the x_i conditions follows as before, since if some vertex of E is required on the outer face, then all of E is.

Sufficiency. The sufficiency proof is exactly that of Theorem 6.7. This is because the various constructions are not dependent on u differing from s. □

THEOREM 6.9. *Let G be a biconnected DAG with a single source s and let $X = \{x_i\} \subseteq V(G)$ be a set of vertices. Let $\{u,v\}$ be a separation pair of G where $u = s$, E be a 3-connected component of G with respect to $\{u,v\}$, and F be the union of all other components of G with respect to $\{u,v\}$. If it is not true that*

$E^* = (E \cup F^*$-marker) admits an upward plane drawing with w_t (if it exists; otherwise the edge (u,v)) and all vertices of X in E on the outer face,

where

$$F^*\text{-}marker = \begin{cases} M_t & \text{if } v \text{ is a source in } F, \\ M_{uv} & \text{otherwise,} \end{cases}$$

then G admits an upward plane drawing with all vertices of X on the outer face iff

(i) *there is no $x \in X$ contained in F,*

(ii) *$F' = (F \cup E$-marker) admits an upward plane drawing with w_t (if it exists; otherwise the edge (u,v)) on the outer face, and*

(iii) *$E' = (E \cup F$-marker) admits an upward plane drawing with all $x \in X$ on the outer face,*

where

$$E\text{-}marker = \begin{cases} M_t & \text{if } v \text{ is a source in } E, \\ M_{uv} & \text{otherwise,} \end{cases}$$

and

$$F\text{-}marker = \begin{cases} M_t & \text{if } v \text{ is a source in } F, \\ M_{uv} & \text{if } v \text{ is a sink in } F, \\ M_{uvt} & \text{otherwise.} \end{cases}$$

Proof (*outline*). Since E has no upward plane drawing with u and v both appropriately on the outer face, by Lemma 6.2, the only way G could be upward planar is if F can be embedded within an internal (hence the new condition (i)) face of E. Thus the outer face of G is fixed as being some face of the drawing of E' not containing v. It remains to ensure that there is some embedding of F which will fit the structural constraints of the shape of a face shared by s and v in the drawing of E. These are exactly the conditions previously required by E for embedding within the drawing of F. The remainder of the proof does not rely on the triconnectedness of either component, and it is similar to the proof of Theorem 6.8 with the roles of E and F reversed. □

6.4. The algorithm. Here we briefly summarize our algorithm and discuss its complexity.

Given DAG G, we first isolate biconnected components using the algorithm discussed in [23]. By Lemma 6.1, these can be tested independently. This step requires $\Theta(n)$ time, plus the time to test each piece independently.

For a biconnected DAG G, we use the triconnected components algorithm of [13] to find a list of separation pairs which breaks G into triconnected components ($\Theta(n)$ time). This list can be rearranged in linear time so that separation pairs involving the source appear last.

For each cut-set $\{u, v\}$ where $u \neq s$, isolate the source component using depth-first search ($\Theta(n)$ time), apply the appropriate theorem (6.5 or 6.7) to add markers and partition X, then recurse. The required time is given by

$$T(n) = T(k) + T(n-k) + O(n) \quad (k \geq 1),$$

which is $O(n^2)$, assuming we can do the base case on the smaller triconnected digraphs in $O(n^2)$ time.

Given a biconnected G with cut-set $\{s, v\}$, isolate a triconnected component (again with DFS) as E, and test, using the algorithm of §4, to see if has an upward plane embedding as specified in Theorem 6.8; this requires only $O(n)$ time, since a triconnected digraph has at most two embeddings with two specified vertices on the outer face. If E passes, continue with testing F as per Theorem 6.8. Otherwise, the operation is absorbed into the $O(n)$ term in the recurrence above, and we apply Theorem 6.9, forcing F to have u and v on the outer face. The previous recurrence and bound apply to this step.

Given a triconnected G, simply apply the algorithm of §4 to all possible outer faces and test for upward planarity in $O(n^2)$ time.

If we wish to output our G embedded into a planar s-t digraph (for drawing), we must find the necessary set of augmenting edges. This can be done by simply running the final upward planar representation through the algorithm of §4.

The entire algorithm operates in $O(n^2)$ time, and correctness follows from the cited results. We have the following theorem.

THEOREM 6.10. *Testing a single-source acyclic digraph for upward planarity, and outputting an upward planar representation when it exists, can be done in* $O(n^2)$ *time.*

7. Conclusions and further work. We have given a linear-time algorithm to test whether a given single-source digraph has an upward plane drawing strongly equivalent to a given plane drawing and to give a representation for this drawing if it exists. This provides, in combination with the decomposition results of §§5 and 6, an efficient $O(n^2)$ algorithm to test upward planarity of an arbitrary single-source digraph. We have also given a combinatorial characterization of single-source upward planar digraphs which provides new insights into their structure.

A lower bound for the single-source upward planarity problem is not known, although we believe that it may be possible to perform the entire test in subquadratic (perhaps linear) time. An obvious extension of this work would be to find such an algorithm or prove a lower bound.

This paper has concentrated on the issues of efficiently testing for an upward plane drawing. However, with the planar representation which results, we can augment the digraph to a planar s-t digraph with our algorithm of §4, then apply the algorithm of Di Battista and Tamassia to generate an actual drawing in $O(n)$ arithmetic steps, or $O(n \log n)$ arithmetic steps for a straight-line drawing [7]. Since an actual drawing must specify physical coordinates for the vertices, it becomes relevant to ask how big the integer grid must be or, equivalently, how much real precision is required. If bends are allowed, an $O(n)$-by-$O(n)$ grid suffices [7], similar to the case of undirected planar graphs [4], [20]. There is no upper bound known on the worst-case size requirement for straight-line upward plane drawings, but Di Battista, Tamassia, and Tollis [8] have exhibited a class of upward planar digraphs requiring an $\Omega(2^n)$-sized integer grid. Thus, any straight-line drawing algorithm is output sensitive—individual coordinates could require $\Omega(n)$ bits, causing the output size to dominate the arithmetic time. Without an upper bound on the required area, it is not known if the drawing algorithm remains polynomial time—any digraph requiring doubly exponential coordinate size would then need exponential time. It would be interesting to characterize some classes of digraphs which permit straight-line upward plane drawings on a polynomially sized grid. Guaranteeing minimum area in all cases is, however, NP-hard [9].

The more general problem, testing upward planarity of an arbitrary acyclic digraph, has recently been shown to be NP-complete [12]. Another recent development by Bertolazzi and Di Battista [1] shows how to efficiently test a triconnected (multisource, multisink) DAG for upward planarity, a more general analogue of the result in §4.

REFERENCES

[1] P. Bertolazzi and G. Di Battista, *On upward drawing testing of triconnected digraphs*, in Proc. 7th ACM Symposium on Computational Geometry, 1991, pp. 272–280, Tech. report RAP.18.90, Dipartimento di Informatica e Sistemistica, Università degli Studi di Roma "La Sapienza," 1991.

[2] J. A. Bondy and U. S. R. Murty, *Graph Theory with Applications*, MacMillian, New York, 1976.

[3] K. S. Booth and G. S. Lueker, *Testing the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms*, J. Comput. System Sci., 13 (1976), pp. 335–379.

[4] H. de Fraysseix, J. Pach, and R. Pollack, *Small sets supporting Fáry embeddings of planar graphs*, in Proc. 20th ACM Symposium on the Theory of Computing (STOC), 1988, pp. 426–433.

[5] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis, *Algorithms for drawing graphs: An annotated bibliography*, preprint, Department of Computer Science, Brown University, Providence, RI, 1993; ongoing version available via anonymous `ftp` from `ftp.cs.brown.edu:pub/papers/compgeo/`, `gdbiblio.tex.Z`, and `gdbiblio.ps.Z`.

[6] G. Di Battista, W. Liu, and I. Rival, *Bipartite graphs, upward drawings, and planarity*, Inform. Process. Lett., 36 (1990), pp. 317–322.

[7] G. Di Battista and R. Tamassia, *Algorithms for plane representations of acyclic digraphs*, Theoret. Comput. Sci., 61 (1988), pp. 175–178.

[8] 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.

[9] 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.

[10] I. Fáry, *On straight line representations of planar graphs*, Acta. Sci. Math. (Szeged), 11 (1948), pp. 229–233.

[11] I. S. Filotti, G. L. Miller, and J. Reif, *On determining the genus of a graph in $O(v^{O(g)})$ steps*, in Proc. 11th ACM Symposium on the Theory of Computing (STOC), 1979, pp. 27–37.

[12] A. Garg and R. Tamassia, *On the computational complexity of upward and rectilinear planarity testing*, Tech. report CS-94-10, Department of Computer Science, Brown University, Providence, RI, 1994.

[13] J. Hopcroft and R. E. Tarjan, *Dividing a graph into triconnected components*, SIAM J. Comput., 2 (1972), pp. 135–158.

[14] J. Hopcroft and R. E. Tarjan, *Efficient planarity testing*, J. Assoc. Comput. Mach., 21 (1974), pp. 549–568.

[15] M. D. Hutton, *Upward planar drawing of single source acyclic digraphs*, Master's thesis, University of Waterloo, Waterloo, ON, Canada, 1990.

[16] M. D. Hutton and A. Lubiw, *Upward planar drawing of single-source acyclic digraphs*, in Proc. 2nd ACM–SIAM Symposium on Discrete Algorithms (SODA), 1991, pp. 203–211.

[17] D. Kelly, *Fundamentals of planar ordered sets*, Discrete Math., 63 (1987), pp. 197–216.

[18] D. Kelly and I. Rival, *Planar lattices*, Canad. J. Math., 27 (1975), pp. 636–665.

[19] C. R. Platt, *Planar lattices and planar graphs*, J. Combin. Theory Ser. B, 21 (1976), pp. 30–39.

[20] W. Schnyder, *Embedding planar graphs on the grid*, in Proc. 1st ACM–SIAM Symposium on Discrete Algorithms (SODA), 1990, pp. 138–148.

[21] S. K. Stein, *Convex maps*, Proc. Amer. Math. Soc., 2 (1951), pp. 464–466.

[22] E. Steinitz and H. Rademacher, *Vorlesungen über die Theorie de Polyeder*, Julius Springer, Berlin, 1934.

[23] R. E. Tarjan, *Depth-first search and linear graph algorithms*, SIAM J. Comput., 1 (1972), pp. 145–159.

[24] C. Thomassen, *Planar acyclic oriented graphs*, Order, 5 (1989), pp. 349–361.

[25] W. T. Tutte, *Convex representations of graphs*, Proc. London Math. Soc., 10 (1960), pp. 304–320.

[26] K. Wagner, *Bemerkungen zum vierfarbenproblem*, Jahresber. Deutsch. Math.-Verein., 46 (1936), pp. 26–32.

[27] W. T. Trotter, Ed., *Planar Graphs*, DIMACS Series in Discrete Mathematics and Computer Science, Vol. 9, American Mathematical Society, Providence, RI, 1993, pp. 41–57.

SIAM J. COMPUT.
Vol. 25, No. 2, pp. 312–339, April 1996

006

AN EFFICIENT PARALLEL ALGORITHM FOR THE GENERAL PLANAR MONOTONE CIRCUIT VALUE PROBLEM*

VIJAYA RAMACHANDRAN† AND HONGHUA YANG†

Abstract. A planar monotone circuit (PMC) is a Boolean circuit that can be embedded in the plane and that contains only AND and OR gates. Goldschlager, Dymond, Cook, and others have developed NC^2 algorithms to evaluate a special layered form of a PMC. These algorithms require a large number of processors ($\Omega(n^6)$, where n is the size of the input circuit). Yang and, more recently, Delcher and Kosaraju have given NC algorithms for the general planar monotone circuit value problem. These algorithms use at least as many processors as the algorithms for the layered case.

This paper gives an efficient parallel algorithm that evaluates a general PMC of size n in polylog time using only a linear number of processors on an exclusive read exclusive write parameter random-access machine (EREW PRAM). This parallel algorithm is the best possible to within a polylog factor and is a substantial improvement over the earlier algorithms for the problem. The algorithm uses several novel techniques to perform the evaluation, including the use of the dual of the plane embedding of the circuit to determine the propagation of values within the circuit.

Key words. circuit value problem, planar monotone circuit, plane graph, dual graph, parallel algorithm, EREW PRAM

AMS subject classifications. 68Q10, 68Q15, 68Q20, 68Q22, 68Q25, 68R10, 05C10

1. Introduction.

A *Boolean circuit* is a directed network of AND, OR, and NOT gates whose wires do not form directed cycles. The problem of evaluating a Boolean circuit, given the values of its inputs, is called the *circuit value problem* (CVP). This is a central problem in the area of algorithms and complexity. Ladner [16] has shown that CVP is P-complete under log space reductions. Some special cases of CVP have been studied, among which the *monotone circuit value problem*, where the Boolean circuit has only AND and OR gates, and the *planar circuit value problem*, where the Boolean circuit has a plane embedding, have been shown to be P-complete by Goldschlager [9].

A *planar monotone circuit* (PMC) is a Boolean circuit that is both planar and monotone. One interesting special case of CVP is the *planar monotone circuit value problem* (PMCVP), which is the problem of evaluating a PMC. In this paper, we give an efficient parallel algorithm for the PMCVP that runs in polylog time using a linear number of processors. The parallel computation model we use here is the exclusive read exclusive write parameter random-access machine (EREW PRAM) model [14].

Here is a summary of earlier results for the PMCVP. Goldschlager [7], [8], Dymond and Cook [4], and Mayr [17] have shown that the problem of evaluating a special layered form of PMC is in NC^2. The first NC algorithm for the general PMCVP was given by Yang [24]; this algorithm runs in $O(\log^3 n)$ time on an EREW PRAM and uses the straight-line code parallel evaluation technique of Miller, Ramachandran, and Kaltofen [18]. Recently, Delcher and Kosaraju [3] have given another NC algorithm for the general PMCVP that runs in $O(\log^4 n)$ time using a polynomial number of processors on a concurrent read exclusive write (CREW) PRAM. All of the algorithms

* Received by the editors November 22, 1993; accepted for publication (in revised form) August 29, 1994. This work was supported in part by Texas Advanced Research Projects grant 003658480 and NSF grant CCR 90-23059. An extended abstract of this work appears in [21].

† Department of Computer Sciences, University of Texas at Austin, Austin, TX 78712-1188 (vlr@cs.utexas.edu, yanghh@cs.utexas.edu).

listed above use a large number of processors (at least $\Omega(n^6)$, where n is the size of the input circuit).

In earlier work (Ramachandran and Yang [20]), we gave an $O(\log^2 n)$-time EREW PRAM algorithm using a linear number of processors to evaluate a layered PMC. The algorithm we present in this paper, when restricted to evaluate a layered PMC, works with the same processor-time bounds as the one in [20]. However, it is substantially different in that it works on a plane embedding of the PMC and its dual graph instead of exploiting a nice layered structure as in [20]. In one sense, our algorithm can be considered to be simpler than the one in [20], since our new approach allows us to eliminate some tedious case analysis used in [20]. Our algorithm uses some ideas from [20] as well as from [24] and [3]. In the highest level of our algorithm, we use an approach similar to that used in [3] to transform a general PMC into "face f induced subcircuits" (using the terminology of [24]—these circuits are called "focused circuits" in [3]). These subcircuits are then evaluated using an algorithm to evaluate a "one-input-face PMC." The major contribution of our paper is our efficient parallel algorithm to evaluate a one-input-face PMC, which is a PMC, not necessarily layered, all of whose input nodes are on the boundary of one face.

The rest of this paper is organized as follows. In §3, we present our algorithm to evaluate a one-input-face PMC. The treatment in §3 is self-contained and does not depend on any result in [20]. In §4, we give an algorithm that runs in polylog time using n processors on an EREW PRAM for evaluating a face f induced subcircuit given a special type of an input assignment. This algorithm works by recursively applying the algorithm for evaluating a one-input-face PMC. Finally, in §5, we give an algorithm that runs in polylog time using n processors on an EREW PRAM for solving the general PMCVP by recursively applying the algorithm for evaluating a face induced subcircuit.

Our results are of interest for several reasons. In designing our efficient parallel algorithm for the PMCVP, we have developed a variety of efficient parallel algorithms for processing planar directed acyclic graphs (DAGs), especially the technique of working on the dual of a planar DAG. (Other examples of algorithmic techniques based on the dual of a plane embedding can be found in [22], [13].) These tools are likely to be of use in algorithms for other problems on planar directed graphs. Our results are of interest in the context of parallel complexity, since all of the earlier algorithms for the PMCVP used indirect methods, such as the relationship between sequential space and parallel time [1] or the parallel evaluation of straight-line code [18], to place the problem in NC. By using direct techniques, we are not only able to place the problem in NC but are also able to obtain a very efficient algorithm for its solution. Finally, the evaluation of circuits is a basic and important problem in computer science. Planar circuits occur very naturally in the design of integrated circuits, and the requirement that the circuit be monotone is not a restriction if the inputs are available together with their complements. Thus our efficient parallel algorithm for the evaluation of planar monotone circuits could be of practical importance.

2. Preliminaries.

DEFINITION 2.1. *A* face *of a plane graph $C = (V, E)$ is a maximal portion of the plane for which any two points may be joined by a curve such that each point of the curve neither corresponds to a vertex of C nor lies on any curve corresponding to an edge of C. The* boundary *of a face f in C consists of all those points x corresponding to vertices and edges of C having the property that x can be joined to a point of f by* *a curve, all of whose points different from x belong to f. (By this definition, a single edge in f belongs to the boundary of f.)*

DEFINITION 2.2. *An embedded* planar monotone circuit (PMC) *is a plane DAG $C = (V, E)$, where*

(i) *V is the set of gates (or vertices) in the PMC consisting of input nodes, AND gates, and OR gates,*

(ii) *E is the set of directed wires (or edges) in the PMC,*

(iii) *the fan-in (or in-degree) of an input node is* 0 *and of an AND or OR gate is either* 1 *or* 2, *and C may have input nodes that are in different faces,*

(iv) *the fan-out (or out-degree) of an output gate is* 0, *and of other gates is nonzero, and C may have more than one output gate, but all output gates of C are in the same face.*

In the rest of the paper, whenever we use the term PMC, we should assume that the PMC is given with an embedding. In case an embedding is not given, we can use the algorithm in Ramachandran and Reif [19] to obtain one. We assume that the plane embedding of a PMC C is given by its combinatorial definition: a clockwise cyclic ordering of edges incident on each vertex in C, and a counterclockwise cyclic ordered sequence of vertices and edges $g_0, e_0, g_1, e_1, \ldots, g_{k-1}, e_{k-1}$ on the boundary of a face in C such that for any i, $0 \leq i \leq k-1$, the edges e_{i-1} and e_i are incident on vertex g_i and e_{i-1} appears immediately before e_i in the cyclic ordering of the edges incident on g_i.

DEFINITION 2.3. *A* complete input assignment *to a PMC is an assignment of values* 0 *or* 1 *to all input nodes in the PMC. A* partial input assignment *to a PMC is an assignment of values* 0 *or* 1 *to a subset of the input nodes in the PMC. An input node that is not assigned a value in a partial input assignment has an* unknown *value. (A complete input assignment is a special case of a partial input assignment.)*

DEFINITION 2.4. *The* partial evaluation problem of a PMC *is the problem of evaluating the value of every gate in the PMC that can be evaluated, given a partial input assignment to the PMC. The gates in a PMC that cannot be evaluated under a partial input assignment have* unknown *values. A PMC is* completely evaluated *if the value of every gate in it is either* 0 *or* 1, *and it is* partially evaluated *otherwise. The* PMCVP *is the problem of completely evaluating a PMC, given a complete input assignment to the PMC.*

A one-input-face PMC we define below is a PMC with the following differences: (i) It is a restriction of a PMC in that all of its input nodes are on the boundary of a single face. (ii) It is a generalization of a PMC in that it may contain *pseudowires*, which are wires that carry no value. Our algorithm may need to add pseudowires in a PMC during the computation in §3.2.

DEFINITION 2.5. *A* one-input-face PMC *C is a variant of a PMC with the following properties.*

1. *C is a plane DAG consisting of input nodes, AND gates, and OR gates.*

2. *All input nodes of C are on the boundary of a single face f_I. The in-degree of an input node is* 0 *and of an AND or OR gate is either* 1 *or* 2.

3. *A gate in C with out-degree* 0 *is called an* output gate. *The out-degree of other gates or input nodes is nonzero. C may have more than one output gate, but all output gates of C are on the boundary of a single face f_O.*

4. *If f_I and f_O are identical, then the input nodes and the output gates of C may not interlace, i.e. there exists a part of the boundary of f_I which contains all*

input nodes but no output gates.

5. *Some of the gates in C may contain a single output wire that does not carry any value and that goes into a two-input gate* (*note that such a gate with a single output wire that does not carry any value is* not *an output gate*). *We call a wire that does not carry any value a* pseudowire. *Further, a two-input gate g may receive at most one input from a pseudowire, and the value of g only depends on its nonpseudo input wire*(*s*).

We will give a recursive algorithm in §3 that evaluates a one-input-face PMC of size n in $O(\log^2 n)$ time using n processors on an EREW PRAM, where properties (4) and (5) in Definition 2.5 are needed after the first level of recursion.

DEFINITION 2.6. *Reach*$(i_1, \ldots, i_k)$ *for some input nodes* $i_1, \ldots, i_k$ *in a PMC is the part of the PMC that is reachable from* $i_1, \ldots, i_k$. *Given a subcircuit P of a PMC, Reach*(P) *is defined to be the part of the circuit reachable from the input nodes in P. Induced*$(i_1, \ldots, i_k)$ *for some input nodes* $i_1, \ldots, i_k$ *that are on the boundary of a single face f* (*where* $i_1, \ldots, i_k$ *need not be all the input nodes of f*) *in a PMC C is Reach*$(i_1, \ldots, i_k)$ *augmented with a new input node set V_{IN} and a wire set E_{IN}, which are formed as follows: if a gate* $x \in C \setminus$ *Reach*$(i_1, \ldots, i_k)$ *has some output wires* $(x, y_1), (x, y_2), \ldots, (x, y_l)$, $(l \geq 1)$, *pointing to gates* $y_1, y_2, \ldots, y_l$ *in Reach*$(i_1, \ldots, i_k)$, *then we add a new input node i_x to V_{IN} and wires* $(i_x, y_1), (i_x, y_2), \ldots, (i_x, y_l)$ *to E_{IN}. We call such Induced*$(i_1, \ldots, i_k)$ *a face f* induced (sub)circuit.

It is easy to see that a face f induced circuit is still a PMC. A face f induced circuit C_f is not necessarily a one-input-face PMC, since the newly added input nodes can appear in faces other than f in C_f. But C_f is still simpler than a general PMC in the sense that all gates in C_f except the newly added input nodes are reachable from some input nodes on the boundary of face f, and once the values of the new input nodes are known, C_f can be transformed into a logically equivalent one-input-face PMC. We give an algorithm in §4 that partially evaluates a face induced circuit of size n given a special input assignment, in polylog(n) time using n processors on an EREW PRAM, by recursively calling the algorithm for evaluating a one-input-face PMC. In §5, we give an algorithm that completely evaluates a general PMC of size n in polylog(n) time using n processors on an EREW PRAM, by recursively calling the algorithm for partially evaluating a face induced circuit, given a special input assignment.

3. The one-input-face PMC. We first consider the problem of completely evaluating a one-input-face PMC C given a complete input assignment. This treatment appears in §§3.1–3.3. We then solve the problem of partially evaluating a one-input-face PMC in §3.4. Our approach is to first find a set of gates in C that are guaranteed to have value 1 and then recursively evaluate the remaining smaller unevaluated subcircuits of C. In an earlier paper [20], we had considered a special case of a one-input-face PMC, namely, a layered PMC (as mentioned in the introduction). In a layered PMC, a sequence of gates with value 1 at one layer guarantees that a sequence of gates at the next layer will have value 1. The left and the right boundaries of the gates with value 1 are defined by the starting gate and the ending gate of the sequence at each layer, respectively. In a one-input-face PMC C that does not have the layered property, we do not have such a simple correspondence. In the treatment below, we work with the dual of a plane embedding of C and define the left and right boundaries of the gates with value 1 in a manner that allows us to determine the propagation of the 1 values through the circuit.

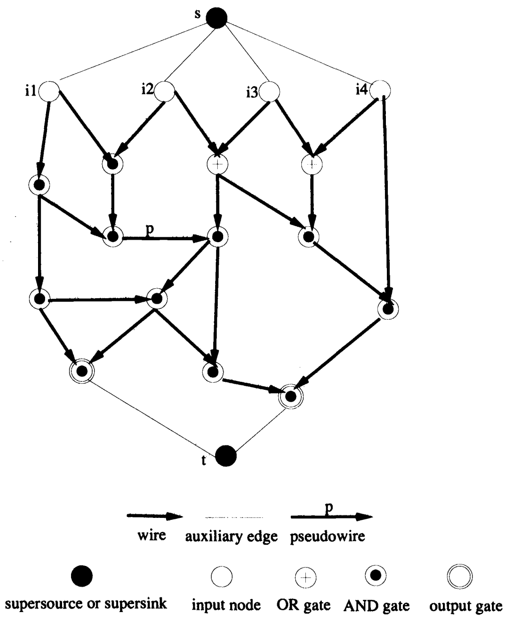


FIG. 1. *C_{aug}: a one-input-face PMC C augmented with a supersource s and a supersink t. Here $f_I = f_O$.*

In §3.1, we will give some definitions and lemmas. In §3.2, we will present our techniques for finding the left and right boundaries of the gates with value 1 and for simplifying the remaining circuit of C. In §3.3, we will give the complete algorithm for evaluating a one-input-face PMC given a complete input assignment and its complexity analysis. In §3.4, we will extend the algorithm to evaluate a one-input-face PMC given a partial input assignment.

Throughout §3, we use C to refer to a one-input-face PMC unless otherwise stated.

3.1. Definitions.

DEFINITION 3.1. *A gate is a* source *of a face f in C if it has two output wires that are on the boundary of f. A gate is a* sink *of a face f in C if it has exactly two input wires and both are on the boundary of f. Let f_I be the face of C whose boundary contains all the input nodes and let f_O be the face of C whose boundary contains all the output gates. C_{aug} is a DAG obtained from C by adding a super source s in f_I and auxiliary wires connecting s to every input node in C and a super sink t in f_O and auxiliary wires connecting every output gate to t (see Fig. 1).*

By Definition 2.5, if f_I and f_O are identical, then the input nodes and the output gates may not interlace in C. Hence C_{aug} is still a plane graph. In the rest of the paper, we assume that C has at least two input nodes. If C has only one input node, then by Definition 2.2, all gates in C have the same value as the only input node, which is a trivial case. The reason that we augment C to C_{aug} is that there is a single source and a single sink in every face of C_{aug} as shown in the following lemma. This property is crucial to many definitions given in this subsection.

Note that not every one-input-face PMC has a downward plane drawing [11] as in Fig. 1 if f_I and f_O are different faces.

LEMMA 3.1. *Every face in C_{aug} has exactly one source and one sink.*

Proof. Every gate in C_{aug} is reachable from s and reachable to t. Since C_{aug} is a DAG, there is at least one source and one sink on the boundary of a face in C_{aug}. Suppose a face f has two sources s_1 and s_2 and therefore two sinks t_1 and t_2. Then we consider the following four directed paths in C_{aug}: the path P_1 from s to s_1, the path P_2 from s to s_2, the path P_3 from t_1 to t, and the path P_4 from t_2 to t. The path P consisting of P_1 and P_2 joins the face f' at s_1 and s_2. The path Q consisting of P_3 and P_4 joins the face f' at t_1 and t_2. But s_1 and s_2 interlace with t_1 and t_2 on the boundary of f. But the two paths P and Q have to be embedded in one side of the boundary of f. This is a contradiction. Hence every face in C_{aug} has exactly one source and exactly one sink. □

The following lemma is needed for Definition 3.3.

LEMMA 3.2. *The output wires of a gate g in C_{aug} are placed consecutively in the cyclic ordering of the wires around g.*

Proof. Let g be a gate in C_{aug} with two input wires i_1 and i_2 and two output wires o_1 and o_2, such that o_1 and o_2 interlace with i_1 and i_2 in the cyclic ordering of the wires around g. Since every gate in C_{aug} is reachable to t, there are two directed paths P_1 and P_2 in C_{aug} from g to t, where P_1 goes through o_1 and P_2 goes through o_2. Let x be the first gate (except g) on P_1 that is also on P_2. Since C_{aug} is acyclic, x must be the first gate (except g) on P_2 that is also on P_1. The subpath of P_1 from g to x and the subpath of P_2 from g to x form an undirected cycle which divides the plane into two parts C_{inside} and $C_{outside}$, where C_{inside} is the part of the plane that is inside the cycle and $C_{outside}$ is the part of the plane that is outside the cycle. Without loss of generality, assume that the supersource s and the input wire i_2 are in $C_{outside}$. Let $i_1 = (g_1, g)$. Then g_1 and i_1 are in C_{inside} and g_1 is not reachable from s since otherwise there would be a directed cycle in C_{aug}. Hence g_1 is reachable from some input nodes in C_{inside} that cannot be reached from s in $C_{outside}$. This is a contradiction. □

DEFINITION 3.2. *The* left (right) input wire *of a two-input gate g is the input wire of g that appears immediately after (before) an output wire of g in the clockwise cyclic ordering of the wires around g.*

Note that the source and the sink of a face f in C_{aug} partition the boundary of f into two disjoint (except at the source and the sink) directed paths.

DEFINITION 3.3. *The path from the source to the sink going through the left input wire of the sink is the* counterclockwise boundary *of the face, and the path from the source to the sink going through the right input wire of the sink is the* clockwise boundary *of the face.*

DEFINITION 3.4. *The* dual digraph $C^*_{aug} = <V^*, E^*>$ *of the plane directed* primal graph $C_{aug} = <V, E>$ *is defined as follows.*

(i) *For each* primal face f *in* C_{aug}, *define a* dual vertex f^* *in* V^*.

(ii) *For each* primal edge e *in* C_{aug} *such that e is on the clockwise boundary of a primal face f_1 and the counterclockwise boundary of another primal face f_2, define a* counterclockwise dual edge $e^{*+} = (f_1^*, f_2^*)$ *in* E^* *and a* clockwise dual edge $e^{*-} = (f_2^*, f_1^*)$ *in* E^*.

Note that in the dual graph C^*_{aug}, we introduce dual edges of both directions, clockwise and counterclockwise, for each edge in the primal graph. The dual graph C^*_{aug} can also be viewed as the result of forming the undirected dual graph of C_{aug} and replacing the dual edges by dual arcs of both directions.

For convenience, for a primal face f, and a primal edge e, we will use f^*, e^{*+},

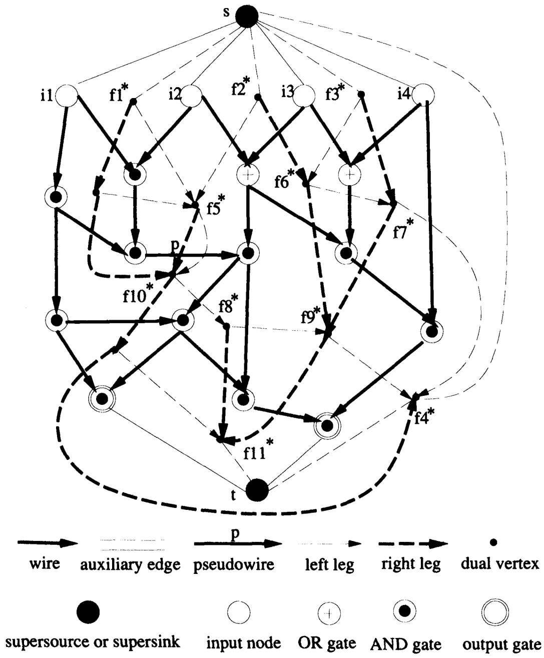


FIG. 2. *C_{aug} and A_{aug}, the auxiliary dual graph. C_{aug} consists of the solid edges. A_{aug} consists of the dashed edges. The two graphs overlap at s and t.*

and e^{*-} to indicate the dual vertex of f, the counterclockwise dual edge of e, and the clockwise dual edge of e, respectively.

In the following definition, we define an auxiliary graph (which can be viewed as a subgraph of C^*_{aug} augmented with s and t) that contains some edges called *left legs* and *right legs.* These edges aid us in defining regions of C_{aug} where value 1 propagates from input nodes. Thus, their definitions are dependent on whether a gate is an AND gate or an OR gate and whether a wire is a pseudowire, since an AND gate has value 1 if both of its inputs have value 1, an OR gate has value 1 if one of its inputs has value 1, and a pseudowire does not pass any value.

DEFINITION 3.5. *The* auxiliary dual graph $A_{aug} = <V_1^*, E_1^*>$ *is defined as follows (see Fig.* 2).

(i) *V_1^* contains the dual vertices of all the primal faces in C_{aug}, together with the supersource s and the supersink t.*

(ii) *E_1^* contains all the dual edges called the* left legs *and the* right legs *defined as follows.*

Let f be a primal face in C_{aug} whose boundary does not contain t, and let g be the sink of f with left input wire w_l and right input wire w_r.

(i) *If g is an OR gate with no pseudo input wire, then the* left leg *and the* right leg *of f^* are w_l^{*-} (i.e., the clockwise dual edge of w_l) and w_r^{*+} (i.e., the counterclockwise dual edge of w_r), respectively.*

(ii) *If g is an AND gate with no pseudo input wire, then the* left leg *and the* right leg *of f^* are w_r^{*+} and w_l^{*-}, respectively.*

(iii) *If w_l is a pseudowire, then both the* left leg *and the* right leg *of f^* are w_l^{*-}.*

(iv) *If w_r is a pseudowire, then both the* left leg *and the* right leg *of f^* are w_r^{*+}.*

Let f be a primal face whose boundary contains t or s in C_{aug}. Then we add an auxiliary edge (f^*, t) or (s, f^*) to E_1^*, respectively.

$A_{aug} \setminus \{s, t\}$ is a subgraph of C_{aug}^* that contains some dual edges of the input wires to the sinks of the faces in C_{aug}. It is easy to see that after being augmented with s and t, A_{aug} is still a plane graph, since the input nodes and the output gates of C do not interlace.

LEMMA 3.3. *If there is an edge $e^* = (f_1^*, f_2^*)$ in A_{aug} which is either a left leg or a right leg, then there is a directed path of length at least* 1 *from the sink s_1 of f_1 to the sink s_2 of f_2 in C_{aug}.*

Proof. By Definition 3.5, s_1 must be on either the clockwise boundary or the counterclockwise boundary of f_2 in C_{aug}. Since a gate (except t) has at most two input wires and therefore can be the sink of at most one face, s_1 cannot be the sink of f_2. Hence there is a directed path of length at least 1 from s_1 to s_2. □

COROLLARY 3.3.1. *A_{aug} is a plane DAG whose only vertex with out-degree* 0 *is t.*

Proof. By Lemma 3.3, A_{aug} is acyclic and hence a DAG. It is obvious that the only vertex in A_{aug} with out-degree 0 is t. □

DEFINITION 3.6. *Two input nodes i_1 and i_2 in C_{aug} are* adjacent *if the wire (s, i_1) and the wire (s, i_2) are adjacent in the cyclic ordering of the wires around s in C_{aug}. Given a complete input assignment to the input nodes of C_{aug}, a* valid base B *is a maximal sequence of adjacent input nodes with value* 1. *The* left (right) bounding face *of a valid base B is the face in C_{aug} whose clockwise (counterclockwise) boundary contains an input node in B, but whose counterclockwise (clockwise) boundary does not (see Fig.* 5*).*

If all input nodes in C_{aug} have value 1, then the left bounding face and the right bounding face are not defined. But this is a trivial case, since we know that all gates of C_{aug} must have value 1.

DEFINITION 3.7. *For a valid base B in C_{aug}, let f_l and f_r be the left and right bounding faces of B, respectively. The* left boundary *and the* right boundary *of B are the two directed paths P_l^* and P_r^*, respectively, in A_{aug}, such that* (1) *P_l^* and P_r^* start from s,* (2) *P_l^* consists of left legs and auxiliary edges and goes through (s, f_l^*),* (3) *P_r^* consists of right legs and auxiliary edges and goes through (s, f_r^*), and* (4) *P_l and P_r end at their first common vertex g^* (g^* could be t) after s (see Figs.* 5 *and* 6*).*

DEFINITION 3.8. *Given a valid base B, the left boundary P_l^* of B and the right boundary P_r^* of B divide the plane into two regions. The region whose counterclockwise boundary is P_l^* and whose clockwise boundary is P_r^* is called the* internal region *of B, and the other region is called the* external region *of B (see Figs.* 5 *and* 6*).*

LEMMA 3.4. *Given a complete input assignment to C_{aug} where there is only one valid base B (all other input nodes have value* 0*), a gate g in C_{aug} evaluates to* 1 *iff g is in the internal region of B.*

Proof. Let us embed A_{aug} and C_{aug} in the plane simultaneously with the same supersource s and the same supersink t (as in Fig. 2) such that the only primal edges in C_{aug} that cross the left and right boundaries P_l^* and P_r^* of B are the input wires to the sinks of some of the primal faces in C_{aug}. (This is provable by Definitions 3.5 and 3.7.) A sink in C_{aug} with a pseudo input wire can have only its pseudo input wire (but not the other input wire) crossing P_l^* or P_r^*. Further, a sink of a primal face f in C_{aug} cannot have its two input wires crossing an edge w_l^* in P_l^* and an edge w_r^* in P_r^*, respectively, since otherwise f^* would be the common starting vertex of both w_l^* and w_r^* and therefore would be a common vertex of P_l^* and P_r^*. However, P_l^* and P_r^*

do not end at f^* since w_l^* is in P_l^* and w_r^* is in P_r^*. This is a contradiction.

Therefore, if we remove all wires that cross P_l^* and P_r^* from C_{aug} and call the resulting graph C'_{aug}, then every gate (except the input nodes) in C'_{aug} still has at least one nonpseudo input wire (by the previous paragraph) and hence can still be reached from some input nodes without going through pseudowires. Further, the gates in the internal (external) region of B in C'_{aug} can be reached only by the input nodes in the internal (external) region. Therefore, if we remove all wires crossing P_l^* and P_r^* from C_{aug}, the gates in the internal region of B will have value 1 and the gates in the external region of B will have value 0. We now show that this is still the case even if we do not remove the wires crossing P_l^* and P_r^* from C_{aug}. By Definition 3.5, a wire in C_{aug} outgoing from a gate in the external region of B and incoming to a gate in the internal region of B is either an input wire to a two-input OR gate or a pseudo input wire to a two-input gate. Hence the gates in the internal region of B in C_{aug} will still have value 1. A primal edge in C_{aug} outgoing from the internal region of B to the external region of B is either an input wire to a two-input AND gate that is in the external region of B or a pseudowire to a two-input gate that is in the external region of B. Hence the gates in the external region of B in C_{aug} will still have value 0. Hence the lemma holds. □

COROLLARY 3.4.1. *Given a complete input assignment to C_{aug}, if a gate g is in the internal region of a valid base in C_{aug}, then g evaluates to* 1.

Proof. The corollary is proved by Lemma 3.4 and the monotonicity of C_{aug}. □

Note that the reverse of Corollary 3.4.1 need not be true if there is more than one valid base in C_{aug}, i.e., some gates outside the internal regions of the valid bases of C_{aug} might also be evaluated to 1. Hence our approach to evaluate a one-input-face PMC C_{aug} is to first find some of the gates that evaluate to 1 based on the internal regions of the valid bases of C_{aug}, remove them from C_{aug}, and repeatedly evaluate the resulting C_{aug}.

3.2. Complete evaluation of a one-input-face PMC. In this subsection, we give an efficient method for computing the left and the right boundaries for all valid bases in C_{aug} simultaneously, given a complete input assignment to C_{aug}. The main idea is to identify for each dual vertex f^* in A_{aug} whether it is on the left (right) boundary of some valid base in C_{aug} (i.e., whether $BOUN_l(f^*)$ or $BOUN_r(f^*)$ defined in Definition 3.10 below is nonempty). Based on this approach, we present a technique to transform the part of C_{aug} that has not been evaluated to 1 into several subcircuits that are one-input-face PMCs with smaller sizes and, more importantly, with geometrically decreasing number of valid bases.

We first define two tree structures that consist of left legs and right legs.

DEFINITION 3.9. *Let V_l (V_r) be the set of vertices of A_{aug} (except s) that are reachable through left (right) legs and auxiliary edges incoming to t from the dual vertex of the left (right) bounding face of some valid base in C. We define T_l^* (T_r^*) to be the subgraph of A_{aug} induced by V_l (V_r).*

For example, Fig. 3 gives T_l^* and T_r^* for the circuit in Fig. 1 with an input assignment $(0, 1, 0, 1)$ to the input nodes i_1, i_2, i_3, i_4.

LEMMA 3.5. *Both T_l^* and T_r^* are convergent trees.*

Proof. By Corollary 3.1.1, A_{aug} is a DAG. By Definition 3.5, there is exactly one left leg and one right leg or one auxiliary edge outgoing from each vertex (except s) in A_{aug}. Hence the lemma holds. □

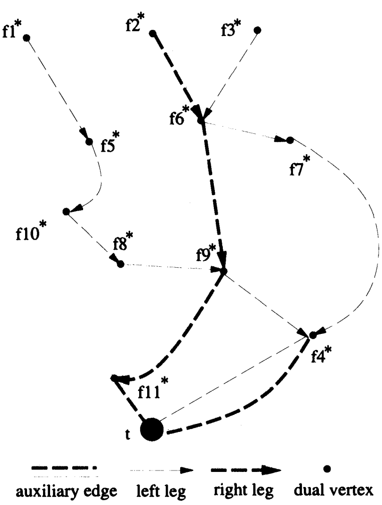


FIG. 3. T_l^* *and* T_r^* *for the circuit in Fig.* 1, *given an input assignment* $(0,1,0,1)$ *to the input nodes* i_1, i_2, i_3, i_4. T_l^* *consists of the light dashed edges.* T_r^* *consists of the dark dashed edges.*

In the following definitions and lemmas, we define $BOUN_l(f^*)$ $(BOUN_r(f^*))$ and related concepts and describe our approach to compute $BOUN_l(f^*)$ $(BOUN_r(f^*))$. Among the sets defined below are two related and similar concepts, $BOUN_l(f^*)$ $(BOUN_r(f^*))$ and $BASE_l(f^*)$ $(BASE_r(f^*))$. These sets are different in that the latter is a superset of the former and is defined to aid the computation of the former.

DEFINITION 3.10. *$PRED_l(f^*)$ $(PRED_r(f^*))$ is the set of the* proper predecessors *of f^* in T_l^* (T_r^*), i.e., the set of dual vertices that can reach f^* through directed paths of length at least* 1 *in T_l^* (T_r^*).*

We associate with each dual vertex f^* in T_l^* (T_r^*) the following sets of valid bases of C_{aug}:

(i) $BASE_l(f^*)$ $(BASE_r(f^*))$ is the set of valid bases B such that the dual vertex of the left (right) bounding face of B is either f^* or in $PRED_l(f^*)$ $(PRED_r(f^*))$. (Informally, $BASE_l(f^*)$ $(BASE_r(f^*))$ is the set of the valid bases in C_{aug} whose left (right) boundaries either contain f^* or a predecessor of f^* in T_l^* (T_r^*).)

(ii) $BOUN_l(f^*)$ $(BOUN_r(f^*))$ is the set of valid bases B whose left (right) boundary contains f^*.

(iii) $JOIN(f^*) = BASE_l(f^*) \cap BASE_r(f^*)$. (Informally, $JOIN(f^*)$ is the set of valid bases whose left and right boundaries either terminate at f^*, or terminate at a predecessor of f^* and the extension of the left and right boundaries rejoin at f^*.)

(iv) $TERM(f^*)$ is the set of valid bases whose left and right boundaries terminate exactly at f^*.

For convenience, if a set for a dual vertex in A_{aug} cannot be defined through Definition 3.10 (e.g., if a dual vertex is not in T_l^*), we assume that it is empty.

Figure 4 illustrates the above definitions. For valid base $B_2 = \{i_4\}$ with left bounding face f_3 and right bounding face f_4 in Fig. 1, the left boundary of B_2 is a subpath of the path from f_3^* to t in T_l^* and the right boundary of B_2 is a subpath of the path from f_4^* to t in T_r^* (in this case, the subpath is the single vertex f_4^*). Notice the difference between $BASE_l(f_4^*) = \{B_1, B_2\}$ and $BOUN_l(f_4^*) = \{B_2\}$, between $BASE_r(t) = \{B_1, B_2\}$ and $BOUN_r(t) = \phi$, and between $JOIN(t) = \{B_1, B_2\}$ and

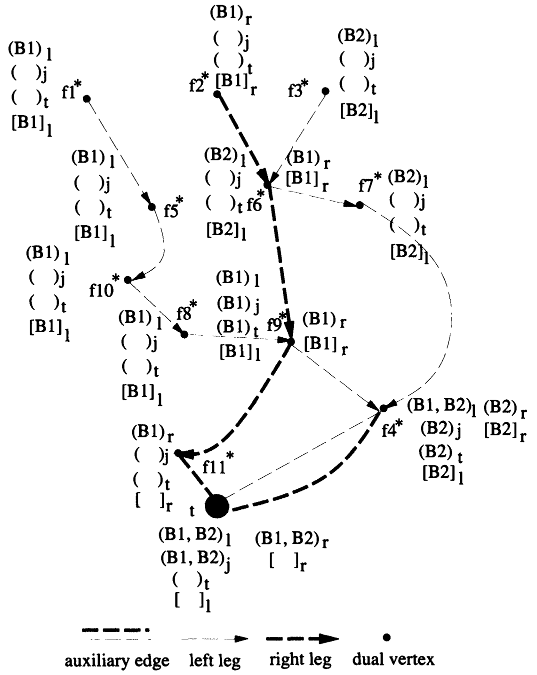


FIG. 4. *Illustrations for the sets $BASE_l$, $BASE_r$, $BOUN_l$, $BOUN_r$, $JOIN$, and $TERM$ on T_l^* and T_r^*. For each node f^* in T_l^*, the contents of $()_l$ denote $BASE_l(f^*)$, the contents of $[]_l$ denote $BOUN_l(f^*)$, the contents of $()_j$ denote $JOIN(f^*)$, and the contents of $()_t$ denote $TERM(f^*)$. For each node f^* in T_r^*, the contents of $()_r$ denote $BASE_r(f^*)$ and the contents of $[]_r$ denote $BOUN_r(f^*)$.*

$TERM(t) = \phi$.

The relations among these sets are summarized in the following lemma. For convenience, we will focus on the sets with index l. The relations among the sets with index r are symmetric.

LEMMA 3.6. *Let f^* be a dual vertex in A_{aug}. Then the following hold:*

1. *$BASE_l(f_p^*) \subseteq BASE_l(f^*)$ for any $f_p^* \in PRED_l(f^*)$.*
2. *$BOUN_l(f^*) \cap BOUN_l(f_1^*) = \phi$ and $BASE_l(f^*) \cap BASE_l(f_1^*) = \phi$ for any f_1^* that does not have predecessor–successor relation with f^* in T_l^*.*
3. *$TERM(f^*) \cap TERM(f_1^*) = \phi$ for any $f_1^* \neq f^*$.*
4. *$TERM(f^*) = JOIN(f^*) \setminus \cup_{f_p^* \in PRED_l(f^*)} JOIN(f_p^*)$.*
5. *$BOUN_l(f^*) = BASE_l(f^*) \setminus \cup_{f_p^* \in PRED_l(f^*)} TERM(f_p^*)$, and $|BOUN_l(f^*)| = |BASE_l(f^*)| - \sum_{f_p^* \in PRED_l(f^*)} |TERM(f_p^*)|$.*

Proof. The correctness of (1)–(3) follows directly from Definition 3.10. The correctness of (4) follows from Definition 3.7, Definition 3.10, (1), and (2). The correctness of (5) follows from Definition 3.10 and (1)–(3). □

Our goal is to identify the gates that are on the left and right boundaries of some valid base. Since a left leg (f_1^*, f_2^*) is on the left boundary of some valid base iff $|BOUN_l(f_1^*)| > 0$ and $|BOUN_l(f_2^*)| > 0$, it suffices to compute $|BOUN_l(f^*)|$ for every f^* in T_l^*. $BASE_l(f^*)$ (and hence $|BASE_l(f^*)|$) can be easily computed using the Euler-tour technique [23] on T_l^* (see Procedure 2 in §3.3 for details). Since

$BASE_l(f^*)$ ($BASE_r(f^*)$) contains valid bases with consecutive labels (modulo the total number of bases) in the total order of the valid bases, it can be described succinctly by a range $[x, y]$, where x and y are the numbers of the first and the last valid bases in $BASE_l(f^*)$ ($BASE_r(f^*)$), respectively. If we can compute $|TERM(f^*)|$ for every f^* that is in both T_l^* and T_r^*, then we can compute $|BOUN_l(f^*)|$ using (5) in Lemma 3.6 and the Euler-tour technique. It remains to compute $|TERM(f^*)|$ for every f^* that is in both T_l^* and T_r^*.

We can try to compute $|TERM(f^*)|$ directly from (4) in Lemma 3.6. However, note that the sets $JOIN(f^*) = BASE_l(f^*) \cap BASE_r(f^*)$ are not necessarily disjoint for different f^* in T_l^* if they have predecessor–successor relation. Instead, we show in the following lemma that they satisfy some important properties, and then in Lemma 3.8, we give a formula to compute $TERM(f^*)$ with disjoint $JOIN(f^*)$ sets.

LEMMA 3.7. (1) *If f_p^* is a predecessor of f^* in both T_l^* and T_r^*, then $JOIN(f_p^*) \subseteq JOIN(f^*)$.*

(2) *Otherwise, $JOIN(f^*) \cap JOIN(f_p^*) = \phi$.*

Proof. (1) By (1) in Lemma 3.6, we have both $BASE_l(f_p^*) \subseteq BASE_l(f^*)$ and $BASE_r(f_p^*) \subseteq BASE_r(f^*)$. Therefore, $JOIN(f_p^*) \subseteq JOIN(f^*)$.

(2) Without lose of generality, assume f^* and f_p^* do not have predecessor–successor relation in T_l^*. Then by (2) in Lemma 3.6, $BASE_l(f^*) \cap BASE_l(f_p^*) = \phi$. Therefore, $JOIN(f_p^*) \cap JOIN(f^*) = \phi$. □

Based on Lemma 3.7, we give the following definition.

DEFINITION 3.11. *For a dual vertex f^* and one of its predecessors f_p^* in T_l^* with $JOIN(f^*) \neq \phi$ and $JOIN(f_p^*) \neq \phi$, $JOIN(f_p^*)$ is* immediately enclosed *by $JOIN(f^*)$, denoted by $JOIN(f_p^*) \subseteq_I JOIN(f^*)$, iff $JOIN(f_p^*) \subseteq JOIN(f^*)$ and there is no dual vertex f_q^* on the directed path from f_p^* to f^* in T_l^* such that $JOIN(f_q^*) \subseteq JOIN(f^*)$.*

LEMMA 3.8. *For a dual vertex f^* that is in both T_l^* and T_r^*,*

1. $TERM(f^*) = JOIN(f^*) \setminus \cup_{f_p^* \in PRED_l(f^*) \wedge JOIN(f_p^*) \subseteq_I JOIN(f^*)} JOIN(f_p^*)$,
2. $|TERM(f^*)| = |JOIN(f^*)| - \sum_{f_p^* \in PRED_l(f^*) \wedge JOIN(f_p^*) \subseteq_I JOIN(f^*)} |JOIN(f_p^*)|$.

Proof. By (4) in Lemma 3.6 and Lemma 3.7, (1) holds immediately. By Definition 3.11, none of the f_p^* in the summation in (2) has predecessor–successor relation. By Lemma 3.7, the sets $JOIN(f_p^*)$ in (2) are disjoint. Hence (2) holds. □

The above lemmas give us the necessary tools to compute $|BOUN_l(f^*)|$ efficiently in parallel. The algorithms that implement the computations in Lemmas 3.6 and 3.8 are given in Procedures 2 and 3 and in the proof of Lemma 3.16 in §3.3. Having computed the left and right boundaries of the valid bases of C_{aug}, our next step is to identify the regions of C_{aug} that consist of gates with value 1. In the following definition, we define a *separating graph* A_{sep}, which is a subgraph of A_{aug} that consists of the left and right boundaries of all the valid bases of C_{aug} and which is used to find the regions of C_{aug} that consists of gates with value 1.

DEFINITION 3.12. (1) *A* separating graph *A_{sep} contains s and the vertices f^* in A_{aug} for which either $BOUN_l(f^*) \neq \phi$ or $BOUN_r(f^*) \neq \phi$. A_{sep} contains t if $BOUN_l(t) \neq \phi$ or $BOUN_r(t) \neq \phi$.*

(2) *An edge (f_1^*, f_2^*) of A_{aug} is an edge in A_{sep} if one of the following three conditions holds*: (a) *$f_1^* = s$ and f_2 is the left or right bounding face of a valid base,* (b) *$BOUN_l(f_1^*) \neq \phi$, $BOUN_l(f_2^*) \neq \phi$ and $BOUN_l(f_1^*) \neq TERM(f_1^*)$, or* (c) *$BOUN_r(f_1^*) \neq \phi$, $BOUN_r(f_2^*) \neq \phi$ and $BOUN_r(f_1^*) \neq TERM(f_1^*)$.*

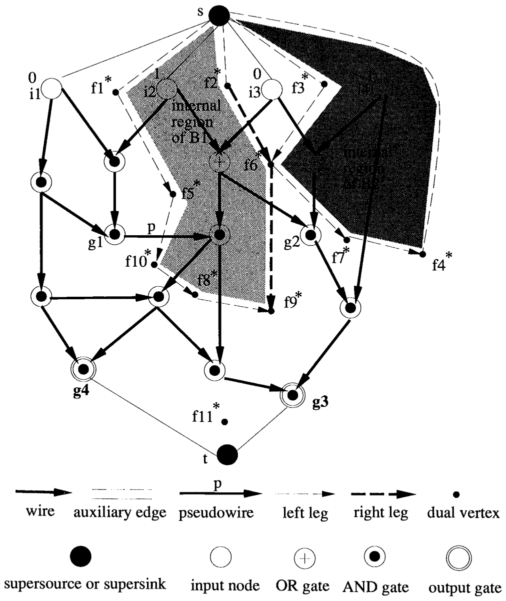


FIG. 5. *C_{aug} and A_{sep} to show the left and right boundaries of B_1 and B_2. C_{aug} consists of the solid edges. The complete input assignment to the input nodes i_1, i_2, i_3, i_4 is $(0,1,0,1)$. $B_1 = \{i_2\}$ and $B_2 = \{i_4\}$ are two valid bases. The left bounding face and the right bounding face of B_1 are f_1 and f_2, respectively. The left boundary of B_1 is the directed path $(s, f_1^*, f_5^*, f_{10}^*, f_8^*, f_9^*)$ and the right boundary of B_1 is the directed path (s, f_2^*, f_6^*, f_9^*). The part of the plane inside the two boundaries is the internal region of B_1 and the part of the plane outside the two boundaries is the external region of B_1. The left bounding face and the right bounding face of B_2 are f_3 and f_4, respectively. The left boundary of B_2 is the directed path $(s, f_3^*, f_6^*, f_7^*, f_4^*)$ and the right boundary of B_2 is the directed path (s, f_4^*). A_{sep} consists of all dashed edges.*

A_{sep} is a subgraph of A_{aug}, which is a subgraph of C^*_{aug} (the dual graph of C_{aug}) augmented with s and t (see Figs. 5 and 6). Hence A_{sep} is a plane graph. Each face in A_{sep} is called a *separating region* of the primal graph C_{aug}. Note that a separating region of C_{aug} either is in the internal region of a valid base or in the external region of every valid base. In the latter case, we call it an *external separating region.*

In the example in Fig. 5, A_{sep} consists of the left and right boundaries of B_1 and B_2. The separating regions of C_{aug} in Fig. 5 are the face in A_{sep} with boundary $(s, f_1^*, f_5^*, f_{10}^*, f_8^*, f_9^*, f_6^*, f_2^*, s)$ (i.e., the internal region of B_1), the face in A_{sep} with boundary $(s, f_3^*, f_6^*, f_7^*, f_4^*, s)$ (i.e., the internal region of B_2), the face in A_{sep} with boundary $(s, f_2^*, f_6^*, f_3^*, s)$, and the face in A_{sep} with boundary $(s, f_1^*, f_5^*, f_{10}^*, f_8^*, f_9^*, f_6^*, f_7^*, f_4^*, s)$.

DEFINITION 3.13. *A wire w is* incoming to (outgoing from) *a subcircuit C' if the head* (*tail*) *of w is a gate in C' but the tail* (*head*) *of w is not.*

LEMMA 3.9. *All incoming wires to an external separating region R in C_{aug} are either wires with value* 1 *or pseudowires.* (*Any input node in R will have value* 0.)

Proof. The wires incoming to R must come from the internal regions of some valid bases, since a wire crosses a boundary of a valid base B either from the internal region to the external region of B, or from the external region to the internal region of B. Since R is not in the internal region of any valid base of C_{aug}, all the wires incoming to R are wires outgoing from the internal regions of some valid bases. By

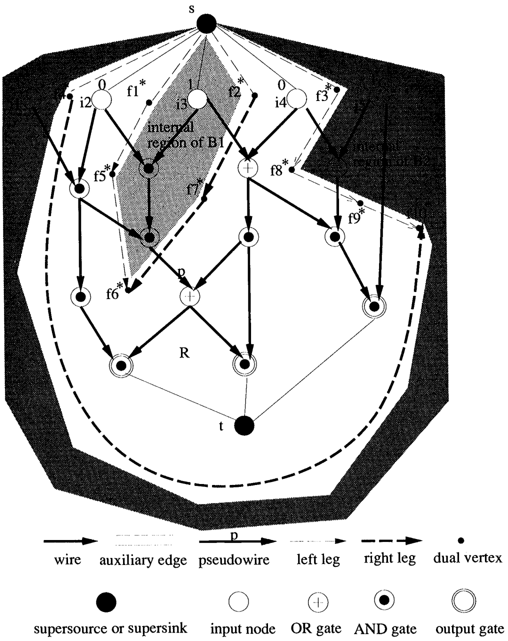


FIG. 6. *C_{aug} and A_{sep} to show the left and right boundaries of B_1 and B_2. C_{aug} consists of the solid edges. A_{sep} consists of the dashed edges. The complete input assignment to the input nodes i_1, i_2, i_3, i_4, i_5 is $(1, 0, 1, 0, 1)$. $B_1 = \{i_3\}$ and $B_2 = \{i_5, i_1\}$ are two valid bases. The left boundary of B_1 is the directed path (s, f_1^*, f_5^*, f_6^*) and the right boundary of B_1 is the directed path (s, f_2^*, f_7^*, f_6^*). The left boundary of B_2 is the directed path $(s, f_3^*, f_8^*, f_9^*, f_{10}^*)$ and the right boundary of B_2 is the directed path (s, f_4^*, f_{10}^*).*

Corollary 3.4.1, the gates in the internal region of any valid base of C_{aug} have value 1. Hence all incoming wires to R are either wires with value 1 or pseudowires. □

Recall that Corollary 3.4.1 states that the gates in the internal region of every valid base of C_{aug} have value 1. Hence the gates in the separating regions that are in the internal region of a valid base have value 1. In the following corollary, we will extend Corollary 3.4.1 to show that in fact the gates in any separating region (including the external separating region) that does not contain an input node with value 0 will have value 1.

COROLLARY 3.9.1. *If a separating region R of C_{aug} does not contain an input node with value* 0, *then all the gates of C_{aug} in R have value* 1.

Proof. If R is in the internal region of a valid base, then by Corollary 3.4.1, the lemma holds. Now we consider an external separating region R that is not in the internal region of any valid base of C. By Lemma 3.9, all incoming wires to R are either wires with value 1 or pseudowires. Since R does not contain an input node with value 0, all the gates in R will have value 1. □

By Corollary 3.9.1, the problem of evaluating the one-input-face PMC C is now reduced to the problem of evaluating each subcircuit of C_{aug} in an external separating region that contains input nodes with value 0, since the gates in other separating regions are known to have value 1. Our next step is to transform these subcircuits into one-input-face PMCs so that we can evaluate these subcircuits recursively. One nontrivial problem with a subcircuit of C_{aug} in a separating region is that the output

```
PROCEDURE 1. Subcircuit transformation
Input: $C_R$, the subcircuit of $C_{aug}$ in an external separating region $R$ containing at least
        one input node with value 0.
Output: $C'_R$, a one-input-face PMC logically equivalent to $C_R$.
0. Initialize $C'_R$ to $C_R$;
1. for each wire $w_i$ in $C_{aug}$ incoming to $C_R$ in parallel do
        let $g$ be the gate in $C_R$ receiving $w_i$ as an input;
        {{note that $g$ must be a two-input AND gate and the sink of a face}}
        (a) if the other input wire of $g$ is in $R$ then remove $w_i$ from $C'_R$;
        (b) else (i.e., the other input wire of $g$ is a wire incoming to $C_R$)
                make $g$ a new input node with value 1 in $C'_R$;
            end {if};
   end {for};
2. for each wire $w_o$ in $C_{aug}$ outgoing from $C_R$ in parallel do
        insert a one-input AND gate $g_{w_o}$ in $w_o$ with $g_{w_o}$ lying inside $R$,
        and remove the part of $w_o$ outgoing from $g_{w_o}$;
        let $w_o^* = (f_1^*, f^*)$ be the dual edge on the boundary of $R$ that crosses $w_o$;
        assume $w_o$ is on the counterclockwise (clockwise) boundary of the face $f$ in $C_{aug}$;
        let $s_f$ be the sink of $f$ in $C_{aug}$;
        let $w^*$ be the other edge connected to $f^*$ on the boundary of $R$ in $A_{sep}$;
        (a) if $w^*$ is an outgoing edge of $f^*$
            then {{$w^*$ crosses either the left input wire $w_l$ or the right input wire $w_r$ of $s_f$}}
                attach an output wire to $g_{w_o}$ in $C'_R$ by adding a pseudowire as follows:
                {{so that $g_{w_o}$ would not be an output gate in $C'_R$, and $C'_R$ is a plane graph}}
                (i) if $w^*$ crosses $w_l$ ($w_r$)
                    then {{$s_f$ must be a two-input AND gate}}
                        connect $g_{w_o}$ and $s_f$ through a pseudowire to replace $w_l$ ($w_r$) in $C'_R$;
                    end {if};
                (ii) if $w^*$ crosses $w_r$ ($w_l$)
                    then make $g_{w_r}$ (i.e., the one-input AND gate inserted in $w_r$ at
                        at the beginning of step 2) a two-input AND gate
                        and connect $g_{w_o}$ to $g_{w_r}$ through a pseudowire in $C'_R$;
                    end {if};
            end {if};
        (b) if $w^*$ is an incoming edge of $f^*$ {{$f^*$ is called a bottom of $R$ in this case}}
            then make $g_{w_o}$ a new output gate in $C'_R$;
            end {if};
   end {for};
end.
```

gates of the subcircuit may interlace with the input nodes of the subcircuit, which makes it impossible to add a supersink to the subcircuit without violating the planarity property. For example, in Fig. 5, after removing the gates in the internal region of B_1 and the internal region of B_2, g_1 will be a new output gate and g_2 will be a new input gate, and the input nodes and the output gates i_1, g_1, g_2, g_3, and g_4 are interlaced with each other in the resulting subcircuit. The following definition and procedure give a method we will apply to solve this problem. This construction uses pseudo wires (defined in part (5) of Definition 2.5).

DEFINITION 3.14. *A circuit C' is* logically equivalent *to a circuit C, if from a partially evaluated C we construct C' (possibly with additional gates) such that for each unevaluated gate g in C, there is a gate in C' with the same value as g.*

The algorithm given in Procedure 1 transforms a subcircuit of C_{aug} in an external separating region that contains at least one input node with value 0 into a logically equivalent one-input-face PMC. Some examples of this transformation are given in Fig. 7. We now show that C'_R constructed by Procedure 1 is a one-input-face PMC that is logically equivalent to C_R.

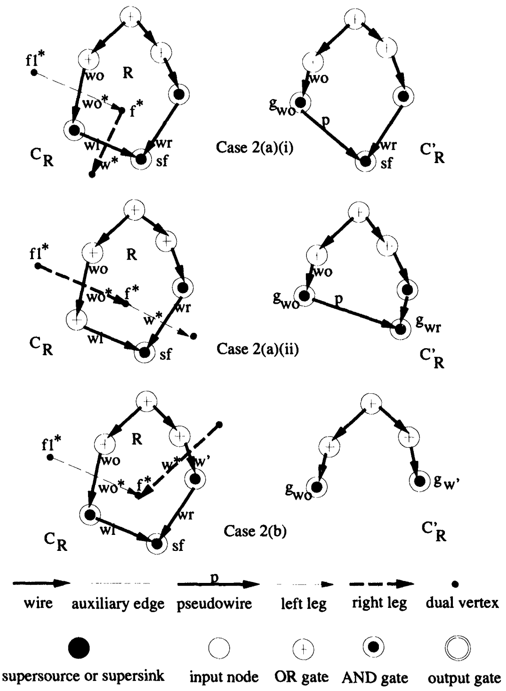


FIG. 7. *The circuit transformation of C_R to C'_R.*

LEMMA 3.10. *C'_R is logically equivalent to C_R.*

Proof. We first show that step 1 in Procedure 1 does not change the value of the gates in C'_R that were originally in C_R. Since C_R is in the external region of every valid base, by Definition 3.5, a wire outgoing from the internal region of some valid base and incoming to R must be either a pseudowire or an input wire to a two-input AND gate whose other input wire is not a pseudowire. By Lemma 3.9, all the incoming wires to R are either wires with value 1 or pseudowires. Hence removing a pseudo input wire or an input wire with value 1 to a two-input AND gate g (whose other input wire is not a pseudowire) in step 1(a) will not change the value of g in C'_R, and the two-input AND gate g in step 1(b) indeed has value 1.

Step 2 in Procedure 1 reduces the number of new output gates in C'_R by adding pseudowires. Steps 2(a)(ii) and 2(b) do not change any input to the gates of C'_R that were originally in C_R. Step 2(a)(i) changes an input to s_f by replacing w_l (w_r) with a pseudowire. However, since w_l (w_r) is an incoming wire to the external separating region R, by the arguments given in the previous paragraph, w_l (w_r) is either a pseudo input wire or an input wire with value 1, s_f is a two-input AND gate, and w_r (w_l) is not a pseudowire. Hence the value of s_f depends only on the value of w_r (w_l) and is the same in both C_R and C'_R. □

LEMMA 3.11. *C'_R is a plane DAG.*

Proof. It is easy to see that C'_R is still a plane graph since the pseudowires introduced in Procedure 1 will not cross any existing wires in C_R.

Suppose there is a directed cycle in C'_R. Then we map a gate g on the cycle to a gate in C_{aug} by the following function f: $f(g) = g$ if g is a gate in C_{aug}; $f(g) = g_2$ if g is not gate in C_{aug} but is a new gate inserted in wire (g_1, g_2) of C_{aug} in step 2

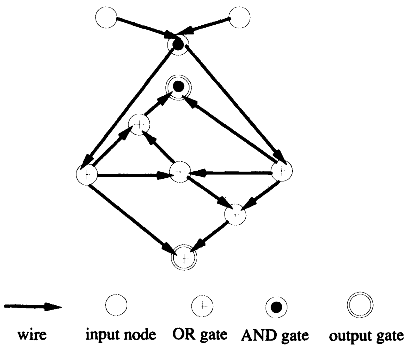


FIG. 8. *An example of a PMC with all input nodes in a single face but output gates in different faces. This PMC cannot be converted into a one-input-face PMC by adding pseudowires to the output gates, since any pseudowire added to an output gate will create a directed cycle in this example.*

of Procedure 1. For each edge (g_1, g_2) on the cycle in C'_R, if (g_1, g_2) is not an edge in C_{aug}, then we add a new edge $(f(g_1), f(g_2))$ to C_{aug} and call the augmented graph C'_{aug}. Hence there is a cycle in C'_{aug} containing new edges. We now prove that for each new edge $(f(g_1), f(g_2))$ in C'_{aug}, there is a directed path in C_{aug} from $f(g_1)$ to $f(g_2)$. We consider the following three cases:

(i) *Case* 1. Both g_1 and g_2 are gates in C_{aug}. Then $(f(g_1), f(g_2)) = (g_1, g_2)$, which is a wire in C_{aug}.

(ii) *Case* 2. g_1 is a newly added gate in C'_R, but g_2 is a gate in C_{aug}. Suppose g_1 is inserted in the wire (g_3, g_4) of C_{aug}. Then (g_1, g_2) is a pseudowire added in step 2(a)(i) of Procedure 1 and g_2 is the sink of the face whose boundary contains g_4. Hence there is a directed path from $f(g_1) = g_4$ to $f(g_2) = g_2$ in C_{aug}.

(iii) *Case* 3. g_2 is a newly added gate in C'_R, but g_1 is a gate in C_{aug}. Then g_2 is inserted in the wire $(g_1, f(g_2))$ of C_{aug} in step 2 of Procedure 1. Hence there is a directed path from $f(g_1) = g_1$ to $f(g_2)$ in C_{aug}.

(iv) *Case* 4. Both g_1 and g_2 are newly added gates in C'_R. Then (g_1, g_2) is a pseudo wire added in step 2(a)(ii) of Procedure 1. Suppose g_1 is inserted in the wire (g_3, g_4) of C_{aug} and g_2 is inserted in the wire (g_5, g_6) of C_{aug}. Then (g_5, g_6) is an outgoing edge from C_R, and g_6 is the sink of the face whose boundary contains g_4. Hence there is a directed path from $f(g_1) = g_4$ to $f(g_2) = g_6$ in C_{aug}.

Hence there is a directed cycle in C_{aug}, which contradicts the fact that C_{aug} is a DAG. Hence C'_R is acyclic. □

At this point, one might wonder if it is the case that any PMC whose input nodes are on the boundary of a single face can be converted to a one-input-face PMC by adding pseudowires to the output gates. The example in Fig. 8 shows that this is not always possible when the output gates are on the boundaries of multiple faces. The construction of C'_R exploited some special properties of a one-input-face PMC and its separating regions to guarantee that the result is a DAG, and this is not always the case when the input circuit is not a one-input-face PMC.

We now show that a subcircuit C'_R output by Procedure 1 must have all inputs in one face, all outputs in one face, and no interlacing of inputs and outputs. This will establish that C'_R is a one-input-face PMC.

A dual vertex f^* is a *bottom* of a separating region R if it is the head of two edges (which are dual edges of C_{aug}) on the boundary of R. (See step 2(b) of Procedure 1

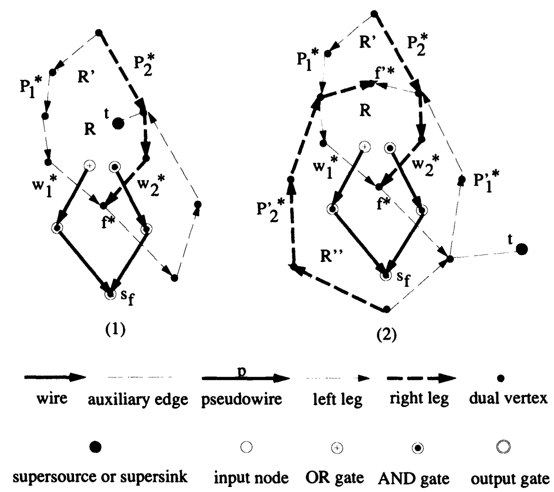


FIG. 9. *Figures for the proof of Lemma* 3.12.

and case 2(b) in Fig. 7.)

LEMMA 3.12. *A separating region R has at most one bottom, and if R has a bottom, then R does not contain the supersink t.*

Proof. Let f^* be a bottom of R and let w_1^* and w_2^* be the two edges incoming to f^*. We find two paths P_1^* and P_2^* in A_{sep} such that (a) P_1^* goes to f^* through w_1^* and P_2^* goes to f^* through w_2^* and (b) P_1^* and P_2^* intersect with each other only at their starting vertices and their ending vertices. Let R' be the region whose counterclockwise boundary is P_1^* and whose clockwise boundary is P_2^*. Then R is inside R', since R is a face in A_{sep}.

We first prove that t is not in R', which implies that t is not in R (see (1) in Fig. 9). Let s_f be the sink of the primal face f. If s_f is t, then we have proved that t is not in R'. Otherwise, since the primal edges of w_1^* and w_2^* are outgoing from R', s_f and its two input wires must be outside of R' (note that the two input wires of s_f cannot be the primal edges of w_1^* and w_2^*, since only the dual edges outgoing from f^* can cross the input wires of s_f). Hence any outgoing edges from f^* in A_{aug} must be outside of R', since they cross the two input wires of s_f. Hence if t were in R', then A_{aug} would have contained a directed cycle, since there is a directed path from f^* to t in A_{aug}. This is a contradiction.

We now prove that R has at most one bottom (see (2) in Fig. 9). Suppose R has another bottom f'^*. Let $w_1'^*$ and $w_2'^*$ be the two edges incoming to f'^*. We find two paths $P_1'^*$ and $P_2'^*$ in A_{sep} such that (a) $P_1'^*$ goes to f'^* through $w_1'^*$ and $P_2'^*$ goes to f'^* through $w_2'^*$ and (b) $P_1'^*$ and $P_2'^*$ intersect with each other only at their starting vertices and their ending vertices. Let R'' be the region whose counterclockwise boundary is $P_1'^*$ and whose clockwise boundary is $P_2'^*$. Then by the proof in the previous paragraph, t is neither in R' nor in R''. But at least one path from f^* or f'^* to t will create a directed cycle in A_{aug}. This is a contradiction. □

COROLLARY 3.12.1. *If C_R contains a bottom, then C'_R does not contain an original output gate of C_{aug}, and C'_R contains at most two newly created output gates, and the two output gates are adjacent to each other on the boundary of a face; if C_R does not contain a bottom, then C'_R does not contain any newly created output gates (C'_R may contain some original output gates of C_{aug}).*

Proof. If C_R contains a bottom, then t is not in C'_R and hence C'_R does not contain an original output gate of C_{aug} (since the auxiliary wires connecting output gates to t do not cross the boundary of R). Further, since C'_R has at most one bottom, at most two new output gates are created in C'_R and they are adjacent to each other on the boundary of a face (see case 2(b) in Fig. 7). If C_R does not contain a bottom, then no new output gates are created in C'_R by the construction in Procedure 1. □

LEMMA 3.13. *All newly created input nodes in C'_R are on the boundary of a single face.*

Proof. After removing all the gates not in R and the wires crossing the boundary of R, all the input nodes in C'_R are on the boundary of a single face, which is the external face of C'_R. Further, the new faces created by the new pseudowires added in step 2 of Procedure 1 do not contain input nodes on their boundaries. □

LEMMA 3.14. *Procedure* 1 *constructs a one-input-face PMC C'_R that is logically equivalent to C_R and runs in $O(1)$ time using a linear number of processors on an EREW PRAM.*

Proof. Lemma 3.13 and Corollary 3.12.1 ensure that the output gates and the input nodes in C'_R do not interlace. By Lemmas 3.10, 3.11, and 3.13 and Corollary 3.12.1, C'_R is a one-input-face PMC that is logically equivalent to C_R.

It is straightforward to see that all steps in Procedure 1 can be implemented in constant time using a linear number of processors. □

We conclude this subsection by showing that a subcircuit C'_R output by Procedure 1 contains at most half the number of valid bases in C_{aug}.

DEFINITION 3.15. *We say two valid bases B and B'* meet *if the right boundary of B and the left boundary of B' have a common vertex. The transitive closure of the meet relation partitions the set of the valid bases in C_{aug} into equivalence classes.*

LEMMA 3.15. *The number of the valid bases in C'_R is at most half of the number of valid bases in C_{aug}.*

Proof. Let g be a newly created input node in C'_R. We say g is a *descendant of* a valid base B of C_{aug} if an original input of g is in the internal region of B. This lemma follows from the following two claims.

CLAIM 1. *Every newly created input node in C'_R is a descendant of at least two distinct valid bases of C_{aug}, and the two valid bases are in the same equivalence class of C_{aug}.*

CLAIM 2. *The newly created input nodes in C'_R that are descendants of the valid bases in the same equivalence class of C_{aug} are in the same valid base in C'_R.*

By Claim 1, a singleton equivalence class of C_{aug} does not generate a new input node with value 1 in C'_R. By Claim 2, an equivalence class of C_{aug} containing at least two valid bases generates at most one valid base in C'_R. Hence the lemma holds.

We first prove Claim 1. Since only the sink of a face can have its input wires crossed by the dual edges in A_{sep}, a newly created input node g must be the sink of a face f in C_{aug}. The two original input wires w_1 and w_2 of g must cross a dual edge w_1^* (which is a left leg) on the left boundary of a valid base and a dual edge w_2^* (which is a right leg) on the right boundary of a valid base, respectively. Since w_1^* and w_2^* are outgoing from the same vertex f^*, they cannot be on the left and right boundaries of the same valid base (since the left and right boundaries end at their first common vertex after s). Therefore, the input wires of g are outgoing from the internal regions of at least two different valid bases (the internal regions of several valid bases may overlap). Further, the two valid bases are in the same equivalence class, since f^* is a

```
ALGORITHM 1. Complete evaluation of a one-input-face PMC
Input: An embedded one-input-face PMC $C$ and a complete input assignment to $C$.
Output: Each gate in $C$ is assigned a value 0 or 1.
1.  if all input nodes in $C$ have value 1
2.  then assign value 1 to all gates in $C$; return;
    else if all input nodes in $C$ have value 0
3.       then assign value 0 to all gates in $C$; return;
         end {if};
    end {if};
4.  Augment $C$ to $C_{aug}$, and construct the auxiliary dual graph $A_{aug}$;
5.  Find the edges in $A_{aug}$ that are on the boundaries of valid bases of $C_{aug}$ (see Procedure 2);
6.  Construct the separating graph $A_{sep}$;
7.  Remove the wires in $C_{aug}$ that cross the boundary edges of $A_{sep}$;
8.  Find the (undirected) connected components in the remaining $C_{aug}$;
9.  for each connected component $C_R$ found in step 8 in parallel do
10.     if $C_R$ does not contain input nodes with value 0
11.     then assign value 1 to all gates in $C_R$;
12.     else transform $C_R$ to $C'_R$ using Procedure 1;
13.          Recursively evaluate $C'_R$;
        end {if};
    end {for};
end.
```

common vertex of the left boundary of one valid base and the right boundary of the other valid base.

We now prove Claim 2. Since the external separating region R contains at least one input node with value 0, the boundary of R must contain the supersource s. Further, s may appear on the boundary of R more than once (see Fig. 6; s appears twice on the boundary of the external separating region R which consists of the part of the plane between the internal region of B_1 and the internal region of B_2). Since multiple appearances of s are possible, if we remove s from the boundary of R, the boundary will be divided into several connected portions, each enclosing a disjoint part of C_{aug} (in Fig. 6, the two disjoint parts are the internal region of B_1 and the internal region of B_2). The valid bases in one part cannot be in the same equivalence class as a valid base in a different part. Let P be a connected portion of the boundary of R after removing S. Let I be the set of all newly created input nodes that are descendants of the valid bases in the equivalence classes enclosed in P. Then the original input wires of the input nodes in I must cross the dual edges in P. Hence the input nodes in I are adjacent on the boundary of a face in C'_R and therefore are in the same valid base in C'_R. □

3.3. An efficient algorithm for the one-input-face PMCVP. Based on the approach we presented in the previous subsection, we give an efficient EREW PRAM algorithm, called Algorithm 1, for evaluating a one-input-face PMC.

The correctness and complexity analysis of Algorithm 1 will be given in Theorem 3.1. All steps in Algorithm 1 are quite straightforward to implement except step 5, which is implemented by Procedure 2. Step 2.5 in Procedure 2 is implemented by Procedure 3, which is similar to a procedure used for the layered PMC in [20].

LEMMA 3.16. *Procedure* 2 (*i.e.*, *step* 5 *in Algorithm* 1) *correctly finds the edges in* A_{aug} *that are on the boundaries of valid bases of* C_{aug} *and runs in* $O(\log n)$ *time using a linear number of processors on an EREW PRAM.*

Proof. The correctness of all steps (except step 2.5) of Procedure 2, which imple-

```
PROCEDURE 2. Finding the edges in $A_{aug}$ that are on the boundaries of valid bases
Input: $C_{aug}$, $A_{aug}$, and a complete input assignment to $C_{aug}$.
Output: The edges of $A_{aug}$ that are on the boundaries of valid bases of $C_{aug}$ are marked.
2.1. Find all the valid bases in $C_{aug}$ and label them in the order of the sequence in which
     they appear on the boundary of the input face of $C_{aug}$;
2.2. Construct $T_l^*$ and $T_r^*$ from $A_{aug}$;
2.3. Compute $BASE_l(f^*)$ and $BASE_r(f^*)$ for each dual vertex $f^*$ in $A_{aug}$;
2.4. Compute $JOIN(f^*) = BASE_l(f^*) \cap BASE_r(f^*)$ for each dual vertex $f^*$ in $A_{aug}$;
2.5. Find the enclosure relation $\subseteq_I$ among the $JOIN(f^*)$ (see Procedure 3);
2.6. Compute $|TERM_l(f^*)|$ and $|TERM_r(f^*)|$ for each dual vertex $f^*$ in $A_{aug}$ using Lemma 3.8;
2.7. Compute $|BOUN_l(f^*)|$ and $|BOUN_r(f^*)|$ for each dual vertex $f^*$ in $A_{aug}$ using Lemma 3.6;
2.8. Mark all dual edges $(f^*, g^*)$ in $A_{aug}$
     with $|BOUN_l(f^*)| > 0$ and $|BOUN_l(f^*)| > |TERM(f^*)|$
     or with $|BOUN_r(f^*)| > 0$ and $|BOUN_r(f^*)| > |TERM(f^*)|$;
end.
```

```
PROCEDURE 3. Finding the enclosure relation $\subseteq_I$ for the $JOIN(f^*)$
Input: $T_l^*$ and $JOIN(f^*)$ for each dual vertex $f^*$ on $T_l^*$.
Output: The enclosure forest $EF^*$ such that a dual vertex $f_p^*$ is the immediate predecessor of
        a vertex $f^*$ in $EF^*$ iff $JOIN(f_p^*) \subseteq_I JOIN(f^*)$.
3.1. for each vertex $f^*$ with nonempty $JOIN(f^*)$ and with the length of the longest path from
         a leaf to $f^*$ in $T_l^*$ being $k$ in parallel do
3.2.     Assign two triples $(x,\ -k,\ f^*)$, $(y,\ k,\ f^*)$ for each range $[x, y]$ in $JOIN(f^*)$;
     end {for};
3.3. Sort all triples into nondecreasing order according to the first two elements in a triple;
3.4. for each triple $(x,\ -k,\ f^*)$, where $k \geq 0$ in parallel do
3.5.     Find its previous triple $(n',\ k',\ f'^*)$ in the sorted list;
3.6.     if $(k' < 0)$ and $(f^* \neq f'^*)$ then $f'^*$ is the parent of $f^*$ in $EF^*$;
3.7.     else if $(k' > 0)$ and $(f^* \neq f'^*)$ then $f'^*$ is the left sibling of $f^*$ in $EF^*$; end {if};
         end {if};
     end {for};
3.8. Construct the $EF^*$ from the parent and sibling relations;
end.
```

ments step 5 of Algorithm 1, has been proved in Lemmas 3.6 and 3.8. We now show the correctness of Procedure 3, which implements step 2.5 of Procedure 2. Let f_1^* and f_2^* be two vertices in T_l^* such that the longest paths from a leaf to f_1^* and from a leaf to f_2^* are of length k_1 and k_2, respectively. By Lemma 3.7, f_1^* is a successor of f_2^* in T_l^* and $JOIN(f_1^*) \supseteq JOIN(f_2^*)$ iff $k_1 > k_2$, and for each range $[x_2, y_2]$ of $JOIN(f_2^*)$, there exists a range $[x_1, y_1]$ of $JOIN(f_1^*)$, such that $x_1 \leq x_2 \leq y_2 \leq y_1$ in the cyclic order. $JOIN(f_1^*) \cap JOIN(f_2^*) = \phi$ iff for each range $[x_2, y_2]$ of $JOIN(f_2^*)$ and each range $[x_1, y_1]$ of $JOIN(f_1^*)$, $x_1 \leq y_1 < x_2 \leq y_2$ in the cyclic order. Therefore, if (n', k', f'^*) and $(x, -k, f^*)$ are two consecutive triples in the sorted list, we have the following: (1) if $k' < 0$ and $f^* \neq f'^*$, then $JOIN(f^*) \subseteq_I JOIN(f'^*)$ and f'^* must be the immediate successor of f^* in the EF^*; (2) if $k' > 0$ and $f^* \neq f'^*$, then f^* and f'^* share the common immediate successor in EF^*.

Next, we analyze the time complexity of Procedure 2.

In step 2.2, T_l^* (T_r^*) can be computed using Euler-tour technique as follows. We first remove s and all right (left) legs from A_{aug}. Then the resulting graph A'_{aug} is a tree rooted at t by the uniqueness of left (right) legs. We then mark the leaf nodes of A'_{aug} that are the dual vertices of the left (right) bounding faces of valid bases of C. Finally, we apply the Euler-tour technique to find all the successors of the marked leaf nodes, and the resulting subtree of A'_{aug} is T_l^* (T_r^*).

In step 2.3, since $BASE_l(f^*)$ $(BASE_r(f^*))$ contains valid bases with consecutive labels (modulo the total number of bases) in the total order of the valid bases, it

```
ALGORITHM 2. Partial evaluation of a one-input-face PMC
Input: A one-input-face PMC $C$ and a partial input assignment to $C$.
Output: Each gate in $C$ that can be evaluated is assigned a value 0 or 1.
1. Assign value 1 to all input nodes with unknown value in $C$ and apply Algorithm 1;
2. Let $A$ be the set of the gates assigned value 0 in this solution of step 1;
3. Assign value 0 to all input nodes with unknown value in $C$ and apply Algorithm 1;
4. Let $B$ be the set of the gates assigned value 1 in this solution of step 3;
5. Assign value 0 to all gates in $A$, assign value 1 to all gates in $B$, and assign unknown value
   to the gates of $C$ that are neither in $A$ nor in $B$;
end.
```

can be described succinctly by a range $[l, h]$ where l and h are the numbers of the first and the last valid bases in $BASE_l(f^*)$ $(BASE_r(f^*))$, respectively. $BASE_l(f^*)$ $(BASE_r(f^*))$ can be computed using Euler-tour technique on T_l^* (T_r^*) as follows. We first label each leaf node of T_l^* (T_r^*) with the label of its corresponding valid base. Then for each vertex f^* in T_l^* (T_r^*), we apply the Euler-tour technique to find the smallest label and the largest label among the leaf predecessors of f^* in T_l^* (T_r^*) and assign them to l and h, respectively.

In step 2.4, $JOIN(f^*)$ can be computed from $BASE_l(f^*)$ and $BASE_r(f^*)$ in constant time and be represented by at most two ranges.

Based on the above analysis, we conclude that steps 2.2–2.4 can be implemented in $O(\log n)$ time using a linear number of processors.

Procedure 3 (which implements step 2.5 in Procedure 2) can be implemented in $O(\log n)$ time with a linear number of processors using the parallel merge sort of [2] and the Euler-tour technique.

It is easy to see that all other steps of Procedure 2 can be implemented in $O(\log n)$ time using a linear number of processors by computing prefix sums and applying Euler-tour and tree-evaluation techniques in [6] and [15]. □

THEOREM 3.1. *Algorithm* 1 *correctly solves the complete evaluation problem of a one-input-face PMC, given a complete input assignment, and runs in* $O(\log^2 n)$ *time using n processors on an EREW PRAM, where n is the size of the circuit.*

Proof. Steps 1–4 are quite straightforward. The correctness of step 5 is proved by Lemma 3.16. The correctness of steps 6–13 is proved by Corollary 3.9.1 and Lemma 3.14.

It is straightforward to see that all steps except steps 5, 8, and 13 in Algorithm 1 can be implemented in $O(\log n)$ time using a linear number of processors. Lemma 3.16 shows that step 5 can be implemented in the same time complexity. Step 8 can be implemented in $O(\log n)$ time optimally by applying the algorithm in [5] for finding connected components in a planar undirected graph.

By Lemma 3.15, the number of the recursive levels needed to complete the evaluation is $O(\log n)$. Therefore, the overall time needed by Algorithm 1 is bounded by $O(\log^2 n)$. Further, the total number of gates in all remaining subcircuits C_R in step 12 in Algorithm 1 is less than the number of gates in the original C_{aug}, since for each newly inserted gate in C_R, there is a unique gate in the internal region of a valid base being removed. Therefore, the processor bound holds. □

3.4. Partial evaluation of a one-input-face PMC. We extend Algorithm 1 to solve the partial evaluation problem of a one-input-face PMC in Algorithm 2.

THEOREM 3.2. *Algorithm* 2 *correctly solves the partial evaluation problem of a one-input-face PMC, given a partial input assignment, and runs in* $O(\log^2 n)$ *time* *using n processors on an EREW PRAM, where n is the size of the circuit.*

Proof. By the monotonicity of the circuit, A is a subset of the gates that should be evaluated to 0 in the partial evaluation of C, and B is a subset of the gates that should be evaluated to 1 in the partial evaluation of C. Further, we now show that a gate g of C that is neither in A nor in B should have unknown values in the partial evaluation of C. Suppose that this is not the case. Let g be a gate that should be evaluated to 0 (1) in the partial evaluation of C, and let g be in neither A nor B. Then g evaluates to 0 (1) under every possible input assignment to the input nodes with unknown values in C. In particular, g has value 0 (1) when all input nodes with unknown values are assigned value 1 (0), which means g is in A (B). This is a contradiction. Therefore, Algorithm 2 is correct.

It is easy to see that the time complexity of Algorithm 2 is the same as that of Algorithm 1, since it is dominated by the two calls on Algorithm 1. □

4. The face induced PMC. In this section, we consider a face f induced circuit C_f, which is defined in §2. For convenience, we assume that C_f is embedded with f being the external face. An *f-partial input assignment* to C_f is a partial input assignment where only input nodes in f can have unknown values and the input nodes in faces other than f must have values 0 or 1. The problem of partially evaluating C_f given an f-partial input assignment is called the *f-partial evaluation* of C_f. Algorithm 3 gives our method to perform an f-partial evaluation of C_f. Algorithm 3 is similar to an algorithm in [3] which first layers a face induced circuit (which squares the size of the circuit) and then recursively partitions the circuit at an appropriate layer. Our algorithm performs a more efficient evaluation by working on a face induced circuit directly and partitioning the circuit according to its topological ordering. It then partially evaluates each subcircuit either recursively or using Algorithm 2.

Recall that a *topological ordering* of a digraph is a linear ordering of its vertices such that every edge in the graph points from a lower-numbered vertex to a higher-numbered vertex. It is well known that a digraph has a topological ordering iff it is a DAG. We now prove the correctness of Algorithm 3 and analyze its complexity.

LEMMA 4.1. *Immediately before step 8, every connected subcircuit in P_l and P_h is a face f' induced circuit for some face f' with an f'-partial input assignment.*

Proof. Let us add to C_f a supersource s in face f and a supersink t in the output face of C_f for the purpose of the proof. We connect s to each input node in f with an edge and connect each output gate to t with an edge. The resulting C_f is still a plane graph.

Only input nodes in f can have unknown values in each connected subcircuit in P_l, since no new input nodes are created in P_l. We now show that the output gates in P_l are in the same face. By step 5, every directed path from a gate in P_h to t consists only of gates in P_h. Hence the gates in P_h can be coalesced to t and the resulting C_f is still a plane graph. The wires outgoing from gates in P_l to gates in P_h are now incoming to t. Hence after we cut the wires outgoing from gates in P_l to t and remove t, the output gates of the connected subcircuits in P_l are in a single face, which we call f_1. Hence every connected subcircuit in P_l is still a face f induced circuit with an f-partial input assignment.

P_h is $C \setminus P_l$ plus some new input nodes with unknown values generated in step 6. The output gates in P_h are not changed and hence are still in the same face. The new input nodes with unknown values are in the same face f_1, since all gates in P_l can be coalesced to s. If there are original input nodes in f remaining in P_h (which

ALGORITHM 3. *f-partial evaluation of a face f induced circuit C_f*

```
Input: A face $f$ induced circuit $C_f$ with an $f$-partial input assignment.
Output: The solution of the $f$-partial evaluation problem of $C_f$.
1. if $C_f$ contains only one gate then return the value of the gate end {if};
2. Obtain a topological ordering of the gates in $C_f$;
3. Let $m$ be the total number of the noninput gates in $C_f$;
4. Find $g_1$ such that there are $\lfloor m/2 \rfloor$ noninput gates before $g_1$ in the topological ordering;
5. Partition the gates in $C_f$ into two parts $P_l$ and $P_h$, such that $P_l$ contains $g_1$ and the gates
   before $g_1$ in the ordering and $P_h$ contains the gates after $g_1$ in the ordering, and remove
   the wires of $C_f$ pointing from gates in $P_l$ to gates in $P_h$;
6. for each gate $g$ in $P_h$ in parallel do
       if all input wire(s) of $g$ are removed
       then replace $g$ by an input node with unknown value in $P_h$;
       else if $g$ is a two-input gate and only one input wire of $g$ is removed
               then add an input node $i$ with unknown value and a wire from $i$ to $g$ in $P_h$;
               end {if};
       end {if};
   end {for};
7. Find the (undirected) connected subcircuits in $P_l$ and $P_h$;
8. $f'$-partially evaluate every connected subcircuit in $P_l$ and $P_h$ recursively in parallel, where $f'$
   is the external input face of the subcircuit;
   {{it will be shown below that each such subcircuit is a face $f'$ induced circuit with an
   $f'$-partial assignment}}
9. Remove all gates that are assigned 0 or 1 in step 8 in $P_h$;
10. Assign the output values of $P_l$ to the input nodes of $P_h$;
11. Partially evaluate every connected subcircuit in $P_h$ using Algorithm 2 in parallel;
   {{it will be shown below that each such subcircuit is a one-input-face PMCs}}
end.
```

are the only input nodes in C_f that possibly carry unknown value), then f_1 must be identical to f. Hence every connected subcircuit in P_h is still a face f_1 induced circuit with an f_1-partial input assignment. □

LEMMA 4.2. *Immediately before step* 11, *every connected subcircuit in P_h is a one-input-face PMC.*

Proof. We show that after removing all gates assigned 0 or 1 in P_h in step 9, no new input nodes are generated, i.e., no gate with in-degree 1 or 2 in P_h becomes a gate with in-degree 0. Let g be a gate with in-degree at least 1 in P_h just before step 9. If all gate(s) that provide inputs to g have known values, then the value of g should be evaluated in step 8 and g should be removed in step 9. If all gate(s) that provide inputs to g have unknown values, then the in-degree of g is not changed. If one input of a two-input gate g has unknown value and the other has known value, then the in-degree of g is 1 after step 9. Hence no new input nodes are generated in P_h in step 9. By Lemma 4.1, every connected subcircuit in P_h and P_l in step 8 is a face f' induced circuit for some input face f' with an f' partial input assignment. Therefore, immediately before step 11, the only input nodes left in each connected subcircuit in P_h are the input nodes in f' that carry unknown value. Hence every connected subcircuit in P_h is a one-input-face PMC. □

THEOREM 4.1. *Algorithm* 3 *correctly solves the f-partial evaluation problem of a face f induced circuit C_f and runs in $O(\log^4 n)$ time using n processors on an EREW PRAM, where n is the size of C_f.*

Proof. The correctness of steps 8 and 11 are shown by Lemmas 4.1 and 4.2. It is straightforward to see that other steps in Algorithm 3 are correct.

Step 1 takes constant time. Step 2 can be implemented in $O(\log^3 n)$ time using n processors on an EREW by Theorem 4.1 in Kao and Klein [12]. The connectivity of a

```
ALGORITHM 4. Complete evaluation of a general PMC
Input: A general PMC $C$ with input nodes $i_1, \ldots, i_m$ and a complete input assignment.
Output: Each gate in $C$ is assigned a value 0 or 1.
0. if $C$ contains only one gate then return the value of the gate end {if};
1. Find the smallest $k$, $0 \leq k \leq m$, such that every connected subcircuit in
   $C \setminus \mathit{Reach}(i_1, i_2, \ldots, i_{(k+1)})$ is of size $\leq n/2$ (see Fig. 10);
   Let $P$ be a connected subcircuit of size $> n/2$ in $C \setminus \mathit{Reach}(i_1, i_2, \ldots, i_k)$ when $k \geq 1$;
2. if $k \geq 1$
3. then Recursively solve the complete evaluation problem for the connected subcircuits in
          $C \setminus \mathit{Reach}(P)$ and in $P \setminus \mathit{Reach}(i_{(k+1)})$ (whose sizes are all $\leq n/2$) in parallel
          (see Fig. 10 and Lemmas 5.1 and 5.2 for steps 3–8);
          {{it will be shown that each such subcircuit is a general PMC with a complete input
          assignment}}
4.        Completely evaluate $\mathit{Induced}(i_{(k+1)}) \cap P$ using Algorithm 3;
          {{it will be shown that each such subcircuit is a face induced circuit with a complete
          input assignment}}
          {{now all gates in $P$ and $C \setminus \mathit{Reach}(P)$ are completely evaluated}}
5.        Remove $P$ from $C$, let $o_1, \ldots, o_{m'}$ be the gates of $P$ with wires outgoing to $\mathit{reach}(P)$;
          {{$o_1, \ldots, o_{m'}$ are on the boundary of a single face in $\mathit{reach}(P)$}}
6.        Completely evaluate $\mathit{Induced}(o_1, \ldots, o_{m'})$ (i.e., $\mathit{Reach}(P) \setminus P$) using Algorithm 3;
          {{it will be shown that each such subcircuit is a face induced circuit with a complete
          input assignment}}
7. else Recursively solve the complete evaluation problem for the connected subcircuits in
        $C \setminus \mathit{Reach}(i_1)$ (whose sizes are all $\leq n/2$) in parallel;
8.      Evaluate $\mathit{Induced}(i_1)$ using Algorithm 3;
   end {if};
end.
```

plane undirected graph in steps 8 and 11 can be solved in $O(\log n)$ time using $n/\log n$ processors on an EREW by the algorithm in Gazit [5]. Steps 3–6 and 9–10 can be implemented in $O(\log n)$ time using $n/\log n$ processors. Step 11 takes $O(\log^2 n)$ time using n processors by Theorem 3.2. Let n' be the number of noninput gates in the original C_f. Since the in-degree of each gate in C_f is ≤ 2, we have $n' < n \leq 3n'$. Each of P_h and P_l contains at most $\lceil n'/2 \rceil$ noninput gates and therefore at most $3\lceil n'/2 \rceil$ total gates (including the new input nodes). Let $T(n)$ be the time needed for Algorithm 3 to partially evaluate a circuit with n gates. We have

$$T(3n') \leq T(3\lceil n'/2 \rceil) + O(\log^3 n).$$

Solving the above recurrence equation, we have $T(3n') = O(\log^4 n)$. Hence $T(n) \leq T(3n') = O(\log^4 n)$. □

5. The general PMCVP. In this section, we give in Algorithm 4 our overall algorithm for evaluating a general PMC. This algorithm evaluates a general PMC recursively by decomposing it into smaller PMCs and disjoint face induced subcircuits. The smaller PMCs are evaluated recursively, while each face induced subcircuit is evaluated by Algorithm 3. We then show the correctness and complexity of Algorithm 4 in Lemma 5.1 and Theorem 5.1. A sketch of an algorithm similar to Algorithm 4 is given in [3].

LEMMA 5.1. *Each connected subcircuit in steps* 3 *and* 7 *is a general PMC with a complete input assignment.*

Proof. Since the gates in $\mathit{Reach}(P)$ can be coalesced into a single gate, the output gates in $C \setminus \mathit{Reach}(P)$ are in the same face. Similarly, the output gates in P and $P \setminus \mathit{Reach}(i_{(k+1)})$ are in the same face. The input nodes in $C \setminus \mathit{Reach}(P)$ are

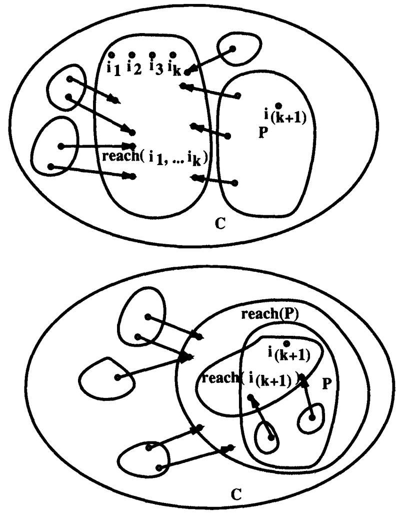


FIG. 10. *A general PMC C of size n, where P is a connected subcircuit of size $> n/2$ in $C \setminus reach(i_1, i_2, \ldots, i_k)$ but $C \setminus reach(i_1, i_2, \ldots, i_{(k+1)})$ does not contain any connected subcircuit of size $> n/2$.*

original input nodes in C. Since there is no wire in C outgoing from a gate in $C \setminus P$ to P, the input nodes in $P \setminus Reach(i_{(k+1)})$ are also original input nodes in C. Hence each connected subcircuit in $C \setminus Reach(P)$ and $P \setminus Reach(i_{(k+1)})$ is a general PMC with a complete input assignment, and can be completely evaluated recursively in step 3. A similar proof holds for step 7. □

LEMMA 5.2. *Each connected subcircuit in steps* 4, 6, *and* 8 *is a face induced circuit with a complete input assignment.*

Proof. The output gates in $Induced(i_{(k+1)}) \cap P$ are in the same face since they are a subset of the output gates in P. $Induced(i_{(k+1)}) \cap P$ are reachable from the original input $i_{(k+1)}$. The other new input nodes in $Induced(i_{(k+1)}) \cap P$ get their value from $P \setminus Reach(i_{(k+1)})$, which is completely evaluated in step 3. Hence $Induced(i_{(k+1)}) \cap P$ is a face f (that contains $i_{(k+1)}$) induced circuit with a complete input assignment and can be completely evaluated using Algorithm 3 in step 4.

The output gates in $Induced(o_1, \ldots, o_{m'})$ (i.e., $Reach(P) \setminus P$) are the output gates in $Reach(P)$, and the output gates in $Reach(P)$ are a subset of the output gates in C and are in the same face. The input nodes $o_1, \ldots, o_{m'}$ in $Reach(P) \setminus P$ are the output gates in P and are in the same face, which we call f_1, and are completely evaluated in steps 3 and 4. All gates in $Reach(P) \setminus P$ are reachable from the input nodes $o_1, \ldots, o_{m'}$ in f_1. The other input gates in $Reach(P) \setminus P$ get values from gates in $C \setminus Reach(P)$, which is completely evaluated in step 3. Hence $Induced(o_1, \ldots, o_{m'})$ (i.e., $Reach(P) \setminus P$) is a face f_1 induced circuit with a complete input assignment and can be completely evaluated using Algorithm 3 in step 6.

A similar proof holds for step 8. □

THEOREM 5.1. *Algorithm* 4 *correctly solves the PMCVP for a general PMC C and runs in $O(\log^6 n)$ time using n processors on an EREW PRAM, where n is the size of the circuit.*

Proof. The correctness of Algorithm 4 has been shown in Lemmas 5.1 and 5.2.

The reachability in steps 1, 3, and 7 can be implemented in $O(\log^4 n)$ time using

n processors on an EREW by the multiple-source reachability algorithm for planar digraphs in Guattery and Miller [10]. The k in step 1 can be found by a binary search. Hence the total time needed in step 1 is $O(\log^5 n)$. The connectivity of a plane undirected graph in steps 1, 3, and 7 can be solved in $O(\log n)$ time using n processors on an EREW by the algorithm in Gazit [5]. By Theorem 4.1, steps 4–6 and 8 can be implemented in $O(\log^4 n)$ time using n processors on an EREW. It is easy to see that the connected subcircuits in steps 3 and 7 are of size $\leq n/2$ and the subcircuits obtained in each step are disjoint. Let $T(n)$ be the time needed for Algorithm 4 to evaluate a PMC with n gates. We have

$$T(n) = T(n/2) + O(log^5 n).$$

Solving the above recurrence equation, we have $T(n) = O(\log^6 n)$. □

Note that the high power in the logarithm for the running time is mainly due to the running time of the reachability algorithms in [10] and [12]. An improvement in the running time of the parallel algorithms for reachability in a plane DAG would imply an improvement in the running time of our algorithm.

REFERENCES

[1] A. Borodin, *On relating time and space to size and depth*, SIAM J. Comput., 6 (1977), pp. 733–744.

[2] R. Cole, *Parallel merge sort*, SIAM J. Comput., 17 (1988), pp. 770–785.

[3] A. L. Delcher and S. R. Kosaraju, *An NC algorithm for evaluating monotone planar circuits*, SIAM J. Comput., 24 (1995), pp. 369–375.

[4] P. W. Dymond and S. A. Cook, *Hardware complexity and parallel computation*, in Proc. 21st IEEE Symposium on Foundations of Computer Science, IEEE Press, Piscataway, NJ, 1980, pp. 360–372.

[5] H. Gazit, *An optimal deterministic* EREW *parallel algorithm for finding connected components in a low genus graph*, in Proc. 5th International Parallel Processing Symposium, IEEE Computer Society Press, Los Alamitos, CA, 1991, pp. 84–90.

[6] A. M. Gibbons and W. Rytter, *An optimal parallel algorithm for dynamic expression evaluation and its applications*, in Proc. Symposium on Foundations of Software Technology and Theoretical Computer Science, Springer-Verlag, Berlin, New York, 1986, pp. 453–469.

[7] L. M. Goldschlager, *A space efficient algorithm for the monotone planar circuit value problem*, Inform. Process. Lett., 10 (1980), pp. 25–27.

[8] ———, *A unified approach to models of synchronous parallel machines*, in Proc. 10th ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1978, pp. 89–94.

[9] ———, *The monotone and planar circuit value problems are* log *space complete for P''*, SIGACT News, 9 (1977), pp. 25–29.

[10] S. Guattery and G. L. Miller, *A contraction procedure for planar directed graphs*, in Proc. 4th ACM Symposium on Parallel Algorithms and Architectures, Association for Computing Machinery, New York, 1992, pp. 431–441.

[11] M. D. Hutton and A. Lubiw, *Upward planar drawing of single source acyclic digraphs*, in Proc. 2nd ACM–SIAM Symposium on Discrete Algorithms, Society for Industrial and Applied Mathematics, Philadelphia, 1991, pp. 203–211.

[12] M. Y. Kao and P. Klein, *Toward overcoming the transitive-closure bottleneck: Efficient parallel algorithms for planar digraphs*, in Proc. 22nd ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1990, pp. 181–192.

[13] M. Y. Kao and G. Shannon, *Local reorientation, global order, and planar topology*, in Proc. 18th ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1986, pp. 160–168.

[14] 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.

[15] S. R. KOSARAJU AND A. L. DELCHER, *Optimal parallel evaluation of tree-structured computations by raking*, in Proc. 3rd Aegean Workshop on Computing, Lecture Notes in Comput. Sci., 319 (1988), pp. 101–110.

[16] R. E. LADNER, *The circuit value problem is* log *space complete for P*, SIGACT News, 7 (1975), pp. 18–20.

[17] E. M. MAYR, *The dynamic tree expression problem*, in Proc. Princeton Workshop on Algorithms, Architecture and Technology Issues for Models of Concurrent Computation, Princeton University Press, Princeton, NJ, 1987, pp. 157–179.

[18] 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.

[19] V. RAMACHANDRAN AND J. H. REIF, *Planarity testing in parallel*, J. Comput. System Sci., 49 (1994), pp. 517–561.

[20] V. RAMACHANDRAN AND H. YANG, *An efficient parallel algorithm for the layered planar monotone circuit value problem*, in Proc. 1st European Symposium on Algorithms, Lecture Notes in Comput. Sci., 726 (1993), pp. 321–332.

[21] ———, *An efficient parallel algorithm for the general planar monotone circuit value problem*, in Proc. 5th ACM–SIAM Symposium on Discrete Algorithms, Society for Industrial and Applied Mathematics, Philadelphia, 1994, pp. 622–631.

[22] ———, *Finding the closed partition of a planar graph*, Algorithmica, 11 (1994), pp. 443–468.

[23] R. E. TARJAN AND U. VISHKIN, *An efficient parallel biconnectivity algorithm*, SIAM J. Comput., 14 (1985), pp. 862–874.

[24] H. YANG, *An NC algorithm for the general planar monotone circuit value problem*, in Proc. 3rd IEEE Symposium on Parallel and Distributed Processing, IEEE Press, Piscataway, NJ, 1991, pp. 196–203.

SIAM J. COMPUT.
Vol. 25, No. 2, pp. 340–354, April 1996

THE BOOLEAN HIERARCHY AND THE POLYNOMIAL HIERARCHY: A CLOSER CONNECTION*

RICHARD CHANG† AND JIM KADIN‡

Abstract. We show that if the Boolean hierarchy collapses to level k, then the polynomial hierarchy collapses to $\mathrm{BH}_3(k)$, where $\mathrm{BH}_3(k)$ is the kth level of the Boolean hierarchy over Σ_2^{P}. This is an improvement over the known results, which show that the polynomial hierarchy would collapse to $\mathrm{P}^{\mathrm{NP}^{\mathrm{NP}}[O(\log n)]}$. This result is significant in two ways. First, the theorem says that a deeper collapse of the Boolean hierarchy implies a deeper collapse of the polynomial hierarchy. Also, this result points to some previously unexplored connections between the Boolean and query hierarchies of Δ_2^{P} and Δ_3^{P}. Namely,

$$\mathrm{BH}(k) = \text{co-BH}(k) \Longrightarrow \mathrm{BH}_3(k) = \text{co-BH}_3(k),$$

$$\mathrm{P}^{\mathrm{NP}\|[k]} = \mathrm{P}^{\mathrm{NP}\|[k+1]} \Longrightarrow \mathrm{P}^{\mathrm{NP}^{\mathrm{NP}}\|[k+1]} = \mathrm{P}^{\mathrm{NP}^{\mathrm{NP}}\|[k+2]}.$$

Key words. polynomial-time hierarchy, Boolean hierarchy, polynomial-time Turing reductions, oracle access, nonuniform algorithms, sparse sets

AMS subject classifications. 68Q15, 03D15, 03D20

1. Introduction.

The Boolean hierarchy (BH) was defined as the closure of NP under Boolean operations and is identical to the difference hierarchy of NP sets [1, 2, 7]. Kadin [3] showed that if the BH collapses at any level, the polynomial-time hierarchy (PH) collapses to $\mathrm{P}^{\mathrm{NP}^{\mathrm{NP}}[O(\log n)]}$, the class of languages in $\mathrm{P}^{\mathrm{NP}^{\mathrm{NP}}}$ that are recognized by deterministic polynomial-time machines that make $O(\log n)$ queries to an $\mathrm{NP}^{\mathrm{NP}}$ oracle. Since the BH is contained in P^{NP}, this result showed that the structure of classes above NP but within P^{NP} is related to the structure of the PH as a whole.

In this paper we extend Kadin's result by showing that if the BH collapses to its kth level (if $\mathrm{BH}(k) = \text{co-BH}(k)$), then the PH collapses to the kth level of the BH within Δ_3^{P} (the difference hierarchy of $\mathrm{NP}^{\mathrm{NP}}$ languages). That is,

$$\text{if } \mathrm{BH}(k) = \text{co-BH}(k), \text{ then } \mathrm{PH} \subseteq \mathrm{BH}_3(k)$$

(see §2 for precise definitions). The kth level of the Δ_3^{P} Boolean hierarchy ($\mathrm{BH}_3(k)$) is contained within $\mathrm{P}^{\mathrm{NP}^{\mathrm{NP}}[\log k+1]}$, the class of languages in $\mathrm{P}^{\mathrm{NP}^{\mathrm{NP}}}$ recognized by machines that make at most $\lceil \log k + 1 \rceil$ queries for all inputs. Therefore, the collapse of the BH implies that the languages within the PH can be recognized by deterministic polynomial-time machines that make a constant number of queries to an $\mathrm{NP}^{\mathrm{NP}}$ oracle, and the deeper the collapse of the BH, the smaller this constant is.

This result also yields two unexpected corollaries:

1. If $\mathrm{BH}(k) = \text{co-BH}(k)$, then $\mathrm{BH}_3(k) = \text{co-BH}_3(k)$.
2. If $\mathrm{P}^{\mathrm{NP}\|[k]} = \mathrm{P}^{\mathrm{NP}\|[k+1]}$, then $\mathrm{P}^{\mathrm{NP}^{\mathrm{NP}}\|[k+1]} = \mathrm{P}^{\mathrm{NP}^{\mathrm{NP}}\|[k+2]}$.

* Received by the editors February 20, 1990; accepted for publication (in revised form) August 21, 1994.

† Computer Science Department, Cornell University, Ithaca, NY 14853. Current address: Department of Computer Science, University of Maryland Baltimore County, Baltimore, MD 21228. The research of this author was supported in part by National Science Foundation research grants DCR-8520597 and CCR-88-23053.

‡ Computer Science Department, University of Maine, Orono, ME 04469. This work was carried out while this author was a visiting scientist at Cornell University.

The first corollary says that the collapse of the BH in Δ_2^{P} implies an identical collapse of the BH in Δ_3^{P}. The second corollary says that the bounded query hierarchies within Δ_2^{P} and Δ_3^{P} are linked. $\mathrm{P}^{\mathrm{NP}\|[k]}$ is the class of languages recognizable by deterministic polynomial-time machines that are allowed to ask k questions in parallel to an oracle from NP (all k queries must be asked at once, so no query can depend on the answers to other queries). $\mathrm{P}^{\mathrm{NP}^{\mathrm{NP}}\|[k]}$ is the class of languages recognizable by deterministic polynomial-time machines that are allowed to ask k questions in parallel to an oracle from $\mathrm{NP}^{\mathrm{NP}}$. If the query hierarchy in Δ_2^{P} collapses, then the query hierarchy in Δ_3^{P} collapses to almost the same level. At first glance, one would think that these corollaries could be proven directly by a straightforward oracle replacement proof. However, the only proof that we know uses a refined version of Kadin's "hard/easy" formulas argument.

The "hard/easy" formulas argument [3, 5], which showed that the collapse of the BH implies the collapse of the PH, went as follows:

- If the BH collapses to its kth level, then the unsatisfiable Boolean formulas of each length n can be partitioned into "easy" and "hard" formulas. The easy formulas can be recognized as unsatisfiable by a particular NP algorithm, and the hard formulas cannot be recognized by this algorithm.
- The hard formulas are key strings, because sequences of at most $k-1$ hard formulas of length n contain enough information to allow an NP machine to recognize all the unsatisfiable formulas of length n.
- A sparse set S was constructed by taking one sequence of hard formulas for each length.
- Since co-NP $\subseteq \mathrm{NP}^S$, the results of Yap [8] imply that PH $\subseteq \Sigma_3^{\mathrm{P}}$.
- By arguing further that $S \in \mathrm{NP}^{\mathrm{NP}}$, it was shown that PH $\subseteq \mathrm{P}^{\mathrm{NP}^{\mathrm{NP}}[O(\log n)]}$.[1]

In this paper we present a deeper analysis of the hard sequences. We show that it is not necessary to choose a particular hard sequence to put into S. In fact, a smaller amount of information, contained in a sparse set T, is enough to allow a Σ_2^{P} machine to recognize Σ_3^{P} languages, i.e.,

$$\mathrm{NP}^{\mathrm{NP}^{\mathrm{NP}}} \subseteq \mathrm{NP}^{T \oplus \mathrm{SAT}}.$$

Since $T \in \mathrm{NP}^{\mathrm{NP}}$ and is almost a tally set (T is a subset of a P-printable set), we are able to show that $\mathrm{NP}^{\mathrm{NP}^{\mathrm{NP}}}$ is contained in the $\mathrm{BH}_3(k)$.

2. Definitions and notation. We assume that the reader is familiar with the classes NPand co-NP, PH, and the NP-complete set SAT.

Notation. For any language L, $L^{\leq n}$ is the set of strings in L of length less than or equal to n. $L^{=n}$ is the set of strings in L of length n.

Notation. We will write π_j for the jth projection function, and $\pi_{(i,j)}$ for the function that selects the ith through jth elements of a k-tuple. For example,

$$\pi_j(\langle x_1, \ldots, x_k\rangle) = x_j,$$
$$\pi_{(1,j)}(\langle x_1, \ldots, x_k\rangle) = \langle x_1, \ldots, x_j\rangle.$$

Notation. We will assume a canonical encoding of the polynomial-time nondeterministic oracle Turing machines, $N_1, N_2, N_3, \ldots$, with effective composition, etc.

[1] Mahaney [6] has found an error in the proof presented in [3] and [5] that the set S is in $\mathrm{NP}^{\mathrm{NP}}$. See [4]. The argument presented in this paper does not use the erroneous reasoning and actually proves a stronger result.

Also, we will assume that all polynomials and running times used in this paper are at least $O(n)$ and are monotone increasing.

DEFINITION. *We write* BH(k) *and* co-BH(k) *for the kth levels of the* BH, *defined as follows:*

$$\mathrm{BH}(1) \overset{\mathrm{def}}{=} \mathrm{NP},$$

$$\mathrm{BH}(k+1) \overset{\mathrm{def}}{=} \{L_1 - L_2 \mid L_1 \in \mathrm{NP} \textit{ and } L_2 \in \mathrm{BH}(k)\},$$

$$\text{co-BH}(k) \overset{\mathrm{def}}{=} \{L \mid \overline{L} \in \mathrm{BH}(k)\}.$$

DEFINITION. *We write* BH$_3$(k) *and* co-BH$_3$(k) *for the kth levels of the* BH *in* Δ_3^{P}*, defined as follows:*

$$\mathrm{BH}_3(1) \overset{\mathrm{def}}{=} \mathrm{NP}^{\mathrm{NP}},$$

$$\mathrm{BH}_3(k+1) \overset{\mathrm{def}}{=} \{L_1 - L_2 \mid L_1 \in \mathrm{NP}^{\mathrm{NP}} \textit{ and } L_2 \in \mathrm{BH}_3(k)\},$$

$$\text{co-BH}_3(k) \overset{\mathrm{def}}{=} \{L \mid \overline{L} \in \mathrm{BH}_3(k)\}.$$

An equivalent way to define the BH *is as follows* [1]:

$$\mathrm{BH}(1) \overset{\mathrm{def}}{=} \mathrm{NP},$$

$$\mathrm{BH}(2k) \overset{\mathrm{def}}{=} \{L \mid L = L' \cap \overline{L_2}, \textit{where } L' \in \mathrm{BH}(2k-1) \textit{and } L_2 \in \mathrm{NP}\},$$

$$\mathrm{BH}(2k+1) \overset{\mathrm{def}}{=} \{L \mid L = L' \cup L_2, \textit{where } L' \in \mathrm{BH}(2k) \textit{and } L_2 \in \mathrm{NP}\},$$

$$\text{co-BH}(k) \overset{\mathrm{def}}{=} \{L \mid \overline{L} \in \mathrm{BH}(k)\}.$$

From this definition, it is not hard to prove that the following languages are complete for the respective levels of the BH under polynomial-time many–one reductions [1].

DEFINITION. *We write* $\mathrm{L}_{\mathrm{BH}(k)}$ *for the canonical complete language for* BH(k) *and* $\mathrm{L}_{\text{co-BH}(k)}$ *for the complete language for* co-BH(k):

$$\mathrm{L}_{\mathrm{BH}(1)} \overset{\mathrm{def}}{=} \mathrm{SAT},$$

$$\mathrm{L}_{\mathrm{BH}(2k)} \overset{\mathrm{def}}{=} \{\langle x_1, \ldots, x_{2k}\rangle \mid \langle x_1, \ldots, x_{2k-1}\rangle \in \mathrm{L}_{\mathrm{BH}(2k-1)} \textit{ and } x_{2k} \in \overline{\mathrm{SAT}}\},$$

$$\mathrm{L}_{\mathrm{BH}(2k+1)} \overset{\mathrm{def}}{=} \{\langle x_1, \ldots, x_{2k+1}\rangle \mid \langle x_1, \ldots, x_{2k}\rangle \in \mathrm{L}_{\mathrm{BH}(2k)} \textit{ or } x_{2k+1} \in \mathrm{SAT}\},$$

$$\mathrm{L}_{\text{co-BH}(1)} \overset{\mathrm{def}}{=} \overline{\mathrm{SAT}},$$

$$\mathrm{L}_{\text{co-BH}(2k)} \overset{\mathrm{def}}{=} \{\langle x_1, \ldots, x_{2k}\rangle \mid \langle x_1, \ldots, x_{2k-1}\rangle \in \mathrm{L}_{\text{co-BH}(2k-1)} \textit{ or } x_{2k} \in \mathrm{SAT}\},$$

$$\mathrm{L}_{\text{co-BH}(2k+1)} \overset{\mathrm{def}}{=} \{\langle x_1, \ldots, x_{2k+1}\rangle \mid \langle x_1, \ldots, x_{2k}\rangle \in \mathrm{L}_{\text{co-BH}(2k)} \textit{ and } x_{2k+1} \in \overline{\mathrm{SAT}}\}.$$

DEFINITION. L_{u_2} *and* L_{u_3} *are the canonical* $\leq_{\mathrm{m}}^{\mathrm{P}}$*-complete languages for* Σ_2^{P} *and* Σ_3^{P}*, respectively:*

$$L_{u_2} \overset{\mathrm{def}}{=} \{(N_j, x, 1^i) \mid N_j^{\mathrm{SAT}}(x) \textit{accepts in } \leq |x| + i \textit{ steps}\},$$

$$L_{u_3} \overset{\mathrm{def}}{=} \{(N_j, x, 1^i) \mid N_j^{L_{u_2}}(x) \textit{accepts in } \leq |x| + i \textit{ steps}\}.$$

3. An example for BH(2) = co-BH(2). In this section we outline the proof of the main theorem for the case $k = 2$. We want to show that if BH(2) = co-BH(2), then $\Sigma_3^{\mathrm{P}} \subseteq \mathrm{BH}_3(2)$. This case contains the spirit of the proof of the main theorem and allows us to illustrate the proof without worrying about even and odd cases or messy indices.

If BH(2) = co-BH(2), then there is a reduction from $\mathrm{L}_{\mathrm{BH}(2)}$ to $\mathrm{L}_{\text{co-BH}(2)}$ via some polynomial-time function h. So, if $h(F_1, F_2) = (G_1, G_2)$, then

$$F_1 \in \mathrm{SAT} \text{ and } F_2 \in \overline{\mathrm{SAT}} \iff G_1 \in \overline{\mathrm{SAT}} \text{ or } G_2 \in \mathrm{SAT}.$$

The key is that h maps a conjunction to a disjunction. Both conditions of the conjunction are met if just one of the disjuncts is met. In the easy case, if G_2 is satisfiable, then F_1 is satisfiable and F_2 *is not satisfiable.* This gives rise to an NP algorithm for recognizing some of $\overline{\mathrm{SAT}}$: Given any formula F_2, guess a formula F_1 with $|F_1| \leq |F_2|$ and accept if $\pi_2 \circ h(F_1, F_2) \in \mathrm{SAT}$.

Formulas that can be recognized as unsatisfiable by this NP algorithm are said to be *easy.* Formally, a Boolean formula F is easy if $\exists F_1$ with $|F_1| \leq |F|$ and $\pi_2 \circ h(F_1, F) \in \mathrm{SAT}$. If all unsatisfiable formulas are easy, then co-NP = NP. So it is likely that there are *hard* unsatisfiable formulas. We say a formula F is *hard* if

1. $F \in \overline{\mathrm{SAT}}$,
2. $\forall F_1$ with $|F_1| \leq |F|$, $\pi_2 \circ h(F_1, F) \in \overline{\mathrm{SAT}}$.

While the set of hard strings is probably not in NP (note that it is in co-NP), an individual hard formula of length m encodes enough information to allow an NP machine to recognize all of $\overline{\mathrm{SAT}}^{\leq m}$. Let F be a hard formula of length m. Suppose F_1 is any formula of length $\leq m$ and $h(F_1, F) = (G_1, G)$. Since F is hard, we know that $F \in \overline{\mathrm{SAT}}$ and $G \in \overline{\mathrm{SAT}}$. Recall that

$$F_1 \in \mathrm{SAT} \text{ and } F \in \overline{\mathrm{SAT}} \iff G_1 \in \overline{\mathrm{SAT}} \text{ or } G \in \mathrm{SAT}.$$

Replacing $F \in \overline{\mathrm{SAT}}$ with "true" and $G \in \mathrm{SAT}$ with "false," we get

$$F_1 \in \mathrm{SAT} \iff G_1 \in \overline{\mathrm{SAT}},$$

or (by negating both sides of the iff)

$$F_1 \in \overline{\mathrm{SAT}} \iff G_1 \in \mathrm{SAT}.$$

So, given the hard string F, an NP machine can recognize if $F_1 \in \overline{\mathrm{SAT}}^{\leq m}$ by computing $G_1 = \pi_1 \circ h(F_1, F)$ and verifying that $G_1 \in \mathrm{SAT}$. In other words, a hard formula of length m and the reduction from BH(2) to co-BH(2) induce a reduction from $\overline{\mathrm{SAT}}^{\leq m}$ to SAT.

The approach taken by Kadin [3] was to encode enough information into a sparse set S so that an NP^S machine could get hold of a hard string of a given length or determine that there was none. Then the NP^S machine could recognize $\overline{\mathrm{SAT}}$, implying that co-NP $\subseteq \mathrm{NP}^S$ and that the PH collapses [8].

In this paper we take a slightly different approach to show that the collapse of the BH implies a deeper collapse of the PH. Rather than constructing a sparse oracle that allows an NP machine to recognize $\overline{\mathrm{SAT}}$, we show that there is a smaller amount of information, essentially a tally set, that allows an $\mathrm{NP}^{\mathrm{NP}}$ machine to recognize the complete language for Σ_3^{P}. For the case where BH(2) = co-BH(2), this information is the tally set

$$T \stackrel{\mathrm{def}}{=} \{1^m \mid \exists \text{ a hard formula of length } m\}.$$

First we show that $\Sigma_3^{\mathrm{P}} \subseteq \mathrm{NP}^{T \oplus \mathrm{NP}}$. Since the set of hard formulas is in co-NP, if we tell an $\mathrm{NP}^{\mathrm{NP}}$ machine that there is a hard formula of length m, it can guess a hard formula and verify with one query that it is hard. With that formula, the $\mathrm{NP}^{\mathrm{NP}}$ machine can produce an NP algorithm that recognizes $\overline{\mathrm{SAT}}^{=m}$. If we tell an $\mathrm{NP}^{\mathrm{NP}}$ machine that there are no hard formulas of length m, then it knows that the "easy" NP algorithm recognizes all of $\overline{\mathrm{SAT}}^{=m}$. In either case, the $\mathrm{NP}^{\mathrm{NP}}$ machine can use an NP algorithm for $\overline{\mathrm{SAT}}^{=m}$ to remove one level of oracle querying from a Σ_3^{P} machine and therefore recognize any Σ_3^{P} language.

Now we show that $\Sigma_3^{\mathrm{P}} \subseteq \mathrm{P}^{\mathrm{NP}^{\mathrm{NP}}[2]}$. Since an $\mathrm{NP}^{\mathrm{NP}}$ machine can guess and verify hard formulas, $T \in \mathrm{NP}^{\mathrm{NP}}$. This implies that a $\mathrm{P}^{\mathrm{NP}^{\mathrm{NP}}}$ machine can tell with one query if there are any hard formulas of a given length. Since this is exactly what an $\mathrm{NP}^{\mathrm{NP}}$ machine needs to recognize a Σ_3^{P} language, the $\mathrm{P}^{\mathrm{NP}^{\mathrm{NP}}}$ machine can pass the information in one more $\mathrm{NP}^{\mathrm{NP}}$ query and therefore recognize a Σ_3^{P} language with only two queries. Hence $\mathrm{BH}(2) = \text{co-BH}(2)$ implies $\Sigma_3^{\mathrm{P}} \subseteq \mathrm{P}^{\mathrm{NP}^{\mathrm{NP}}[2]}$, the class of languages recognizable in deterministic polynomial-time with two queries to $\mathrm{NP}^{\mathrm{NP}}$.

With more work, we can show that Σ_3^{P} is actually contained in the second level of the Δ_3^{P} BH.

4. Main result. We can generalize the analysis of the previous section to higher levels of the BH by replacing the concept of hard formulas with the concept of *hard sequences* of formulas. Just as an individual hard formula could be used with the reduction from BH(2) to co-BH(2) to induce a reduction from $\overline{\mathrm{SAT}}$ to SAT, a hard sequence is a j-tuple that can be used with a $\leq_{\mathrm{m}}^{\mathrm{P}}$-reduction from BH($k$) to co-BH($k$) to define a $\leq_{\mathrm{m}}^{\mathrm{P}}$-reduction from BH($k-j$) to co-BH($k-j$).

DEFINITION. *Suppose* $L_{\mathrm{BH}(k)} \leq_{\mathrm{m}}^{\mathrm{P}} L_{\text{co-BH}(k)}$ *via some polynomial-time function* h. *Then we call* $\langle 1^m, x_1, \ldots, x_j \rangle$ *a* hard sequence *with respect to* h *if* $j = 0$ *or if all of the following hold:*

1. $1 \leq j \leq k-1$.
2. $|x_j| \leq m$.
3. $x_j \in \overline{\mathrm{SAT}}$.
4. $\langle 1^m, x_1, \ldots, x_{j-1} \rangle$ *is a hard sequence with respect to* h.
5. *For all* $y_1, \ldots, y_\ell \in \Sigma^{\leq m}$ *(where* $\ell = k - j$*),*

$$\pi_{\ell+1} \circ h(\langle y_1, \ldots, y_\ell, x_j, \ldots, x_1 \rangle) \in \overline{\mathrm{SAT}}.$$

If $\langle 1^m, x_1, \ldots, x_j \rangle$ is a hard sequence, then we refer to j as the *order* of the sequence and say that it is a hard sequence for length m. Also, we will call a hard sequence *maximal* if it cannot be extended to a hard sequence of higher order. We say that j is the maximum order for length m if there is a hard sequence of order j for length m and there is no hard sequence of order $j+1$ for length m. Finally, when the individual strings $x_1, \ldots, x_j$ are of no importance, we use the shortened notation $\langle 1^m, \vec{x} \rangle$ instead of $\langle 1^m, x_1, \ldots, x_j \rangle$.

Our proof that $\mathrm{BH} \subseteq \mathrm{BH}(k)$ implies $\mathrm{PH} \subseteq \mathrm{BH}_3(k)$ is rather involved. All of our lemmas and theorems start with the assumption that $\mathrm{BH}(k) = \text{co-BH}(k)$ or, in other words, that there exists a function h that is a $\leq_{\mathrm{m}}^{\mathrm{P}}$-reduction from $L_{\mathrm{BH}(k)}$ to $L_{\text{co-BH}(k)}$. First we show that a hard sequence of order j for length m does indeed induce a reduction from $L_{\mathrm{BH}(k-j)}$ to $L_{\text{co-BH}(k-j)}$ (Lemma 4.1). Then we show that a maximal hard sequence for length m induces a polynomial-time reduction from $\overline{\mathrm{SAT}}^{\leq m}$ to SAT (Lemma 4.2). In Lemma 4.3 we argue that given a maximal hard sequence, an NP machine can recognize an initial segment of L_{u_2}, the canonical complete language for

Σ_2^{P}. Lemma 4.4 takes this a step further by showing that given the maximum order of hard sequences for a length, an $\mathrm{NP}^{\mathrm{NP}}$ machine can recognize an initial segment of L_{u_3}, the canonical complete language for Σ_3^{P}. We then define the set T which encodes the orders of hard sequences for each length, and we show that $T \in \mathrm{NP}^{\mathrm{NP}}$ (Lemma 4.5). We put all this analysis together in Theorem 4.6 and show that $\mathrm{BH}(k) = \text{co-BH}(k)$ implies $\mathrm{PH} \subseteq \mathrm{P}^{\mathrm{NP}^{\mathrm{NP}}[k]}$.

Moving toward the Δ_3^{P} BH, we prove that an NP machine can recognize if there is a hard sequence of order j for length m if it is given a hard sequence for a polynomially longer length (Lemma 4.7). In Lemma 4.8 we show that the maximum order of hard sequences for a length is enough information to permit an $\mathrm{NP}^{\mathrm{NP}}$ machine to recognize when a string *is not in* L_{u_3}; that is, the $\mathrm{NP}^{\mathrm{NP}}$ machine can recognize an initial segment of a complete language for Π_3^{P}. Finally, this gives us the machinery to prove our main theorem.

We start by showing that a hard sequence of order j for length m induces a reduction from $\mathrm{L}_{\mathrm{BH}(k-j)}$ to $\mathrm{L}_{\text{co-BH}(k-j)}$ for tuples of strings up to length m.

LEMMA 4.1. *Suppose* $\mathrm{L}_{\mathrm{BH}(k)} \leq_{\mathrm{m}}^{\mathrm{P}} \mathrm{L}_{\text{co-BH}(k)}$ *via some function* h *and* $\langle 1^m, x_1, \ldots, x_j\rangle$ *is a hard sequence with respect to* h. *Then for all* $y_1, \ldots, y_\ell \in \Sigma^{\leq m}$ *(where* $\ell = k - j$*),*

$$\langle y_1, \ldots, y_\ell\rangle \in \mathrm{L}_{\mathrm{BH}(\ell)} \iff \pi_{(1,\ell)} \circ h(\langle y_1, \ldots, y_\ell, x_j, \ldots, x_1\rangle) \in \mathrm{L}_{\text{co-BH}(\ell)}.$$

Proof (by induction on j).

Induction hypothesis $P(j)$. For all $y_1, \ldots, y_{k-j} \in \Sigma^{\leq m}$

$$\langle y_1, \ldots, y_{k-j}\rangle \in \mathrm{L}_{\mathrm{BH}(k-j)} \iff \pi_{(1,k-j)} \circ h(\langle y_1, \ldots, y_{k-j}, x_j, \ldots, x_1\rangle) \in \mathrm{L}_{\text{co-BH}(k-j)}.$$

Base case $P(0)$. By the hypothesis of the lemma, h reduces $\mathrm{L}_{\mathrm{BH}(k)}$ to $\mathrm{L}_{\text{co-BH}(k)}$, so

$$\langle y_1, \ldots, y_k\rangle \in \mathrm{L}_{\mathrm{BH}(k)} \iff h(\langle y_1, \ldots, y_k\rangle) \in \mathrm{L}_{\text{co-BH}(k)}.$$

However, $\pi_{(1,k)} \circ h(\langle y_1, \ldots, y_k\rangle) = h(\langle y_1, \ldots, y_k\rangle)$, so

$$\langle y_1, \ldots, y_k\rangle \in \mathrm{L}_{\mathrm{BH}(k)} \iff \pi_{(1,k)} \circ h(\langle y_1, \ldots, y_k\rangle) \in \mathrm{L}_{\text{co-BH}(k)}.$$

Induction case $P(j+1)$. Suppose $P(j)$ holds. Let $\ell = k - j$. Let $\langle 1^m, x_1, \ldots, x_{j+1}\rangle$ be a hard sequence. By the induction hypothesis, for all $y_1, \ldots, y_{\ell-1} \in \Sigma^{\leq m}$,

$$\langle y_1, \ldots, y_{\ell-1}, x_{j+1}\rangle \in \mathrm{L}_{\mathrm{BH}(\ell)} \iff \pi_{(1,\ell)} \circ h(\langle y_1, \ldots, y_{\ell-1}, x_{j+1}, \ldots, x_1\rangle) \in \mathrm{L}_{\text{co-BH}(\ell)}.$$

If ℓ is even, then by the definitions of $\mathrm{L}_{\mathrm{BH}(\ell)}$ and $\mathrm{L}_{\text{co-BH}(\ell)}$,

(1)
$$\begin{array}{lcl} \langle y_1, \ldots, y_{\ell-1}\rangle \in \mathrm{L}_{\mathrm{BH}(\ell-1)} & & \pi_{(1,\ell-1)} \circ h(\langle y_1, \ldots, y_{\ell-1}, x_{j+1}, \ldots, x_1\rangle) \in \mathrm{L}_{\text{co-BH}(\ell-1)} \\ \quad \text{and } x_{j+1} \in \overline{\mathrm{SAT}} & \iff & \text{or } \pi_\ell \circ h(\langle y_1, \ldots, y_{\ell-1}, x_{j+1}, \ldots, x_1\rangle) \in \mathrm{SAT}. \end{array}$$

If ℓ is odd, then by the definitions of $\mathrm{L}_{\mathrm{BH}(\ell)}$ and $\mathrm{L}_{\text{co-BH}(\ell)}$,

(2)
$$\begin{array}{lcl} \langle y_1, \ldots, y_{\ell-1}\rangle \in \mathrm{L}_{\mathrm{BH}(\ell-1)} & & \pi_{(1,\ell-1)} \circ h(\langle y_1, \ldots, y_{\ell-1}, x_{j+1}, \ldots, x_1\rangle) \in \mathrm{L}_{\text{co-BH}(\ell-1)} \\ \quad \text{or } x_{j+1} \in \mathrm{SAT} & \iff & \text{and } \pi_\ell \circ h(\langle y_1, \ldots, y_{\ell-1}, x_{j+1}, \ldots, x_1\rangle) \in \overline{\mathrm{SAT}} \end{array}$$

or (by negating both sides of the iff)

(3)

$$\begin{array}{ccc} \langle y_1,\dots,y_{\ell-1}\rangle \notin \mathrm{L}_{\mathrm{BH}(\ell-1)} & & \pi_{(1,\ell-1)}{\circ}h(\langle y_1,\dots,y_{\ell-1},x_{j+1},\dots,x_1\rangle) \notin \mathrm{L}_{\text{co-BH}(\ell-1)} \\ \text{and } x_{j+1}\in\overline{\mathrm{SAT}} & \iff & \text{or } \pi_\ell{\circ}h(\langle y_1,\dots,y_{\ell-1},x_{j+1},\dots,x_1\rangle)\in \mathrm{SAT}. \end{array}$$

Since $\langle 1^m, x_1,\dots,x_{j+1}\rangle$ is a hard sequence, we know from parts 3 and 5 of the definition of a hard sequence that $x_{j+1}\in\overline{\mathrm{SAT}}$ and

$$\pi_\ell{\circ}h(\langle y_1,\dots,y_{\ell-1},x_{j+1},\dots,x_1\rangle)\in\overline{\mathrm{SAT}}.$$

Therefore, in equations (1) and (3), the second conjunct on the left side is true and the second disjunct on the right side is false. Hence

$$\langle y_1,\dots,y_{\ell-1}\rangle \in \mathrm{L}_{\mathrm{BH}(\ell-1)} \iff \pi_{(1,\ell-1)}{\circ}h(\langle y_1,\dots,y_{\ell-1},x_{j+1},\dots,x_1\rangle)\in \mathrm{L}_{\text{co-BH}(\ell-1)}.$$

Then, replacing $k-j$ for ℓ, we have

$$\begin{gathered} \langle y_1,\dots,y_{k-(j+1)}\rangle \in \mathrm{L}_{\mathrm{BH}(k-(j+1))} \\ \iff \\ \pi_{(1,k-(j+1))}{\circ}h(\langle y_1,\dots,y_{k-(j+1)},x_{j+1},\dots,x_1\rangle)\in \mathrm{L}_{\text{co-BH}(k-(j+1))}. \end{gathered}$$

So, we have established the induction hypothesis $P(j+1)$. □

Lemma 4.2 shows that a maximal hard sequence for length m induces a polynomial-time reduction from $\overline{\mathrm{SAT}}^{\le m}$ to SAT or, in other words, given a maximal hard sequence for length m, an NP machine can recognize $\overline{\mathrm{SAT}}^{\le m}$.

LEMMA 4.2. *Suppose* $\mathrm{L}_{\mathrm{BH}(k)} \le^{\mathrm{P}}_{\mathrm{m}} \mathrm{L}_{\text{co-BH}(k)}$ *via some function h and* $\langle 1^m, x_1,\dots,x_j\rangle$ *is a maximal hard sequence with respect to h. Then for all* $y\in\Sigma^{\le m}$

$$\begin{gathered} y\in\overline{\mathrm{SAT}} \\ \iff \\ \exists\, y_1,\dots,y_{\ell-1}\in\Sigma^{\le m}\quad \pi_\ell{\circ}h(\langle y_1,\dots,y_{\ell-1},y,x_j,\dots,x_1\rangle)\in\mathrm{SAT} \end{gathered}$$

(where $\ell = k-j$*).*

Proof. If $j=k-1$ ($\langle y_1,\dots,y_{\ell-1}\rangle$ is the empty sequence), then by Lemma 4.1, for all $y\in\Sigma^{\le m}$,

$$y\in\mathrm{BH}(1) \iff \pi_1{\circ}h(\langle y,x_j,\dots,x_1\rangle)\in\text{co-BH}(1).$$

However, BH(1) = SAT and co-BH(1) = $\overline{\mathrm{SAT}}$. So, we have

$$y\in\mathrm{SAT} \iff \pi_1{\circ}h(\langle y,x_j,\dots,x_1\rangle)\in\overline{\mathrm{SAT}}$$

or (by negating both sides of the iff)

$$y\in\overline{\mathrm{SAT}} \iff \pi_1{\circ}h(\langle y,x_j,\dots,x_1\rangle)\in\mathrm{SAT}.$$

Thus, the lemma holds when $j=k-1$ (i.e., when $y_1,\dots,y_{\ell-1}$ is the empty sequence).

Consider the case when $j<k-1$.

($\Rightarrow$) Suppose $y\in\overline{\mathrm{SAT}}$. Since $\langle 1^m,x_1,\dots,x_j\rangle$ is maximal, $\langle 1^m,x_1,\dots,x_j,y\rangle$ is not a hard sequence. However, $j+1\le k-1$, $|y|\le m$, $y\in\overline{\mathrm{SAT}}$, and $\langle 1^m,x_1,\dots,x_j\rangle$

is a hard sequence. So, $\langle 1^m, x_1, \ldots, x_j, y\rangle$ must fail to be a hard sequence by failing to satisfy condition 5 of the definition of hard sequences. Thus

$$\exists\, y_1, \ldots, y_{\ell-1} \in \Sigma^{\leq m} \quad \pi_\ell \circ h(\langle y_1, \ldots, y_{\ell-1}, y, x_j, \ldots, x_1\rangle) \in \mathrm{SAT}.$$

($\Leftarrow$) Suppose that for some $y_1, \ldots, y_{\ell-1} \in \Sigma^{\leq m}$,

$$\pi_\ell \circ h(\langle y_1, \ldots, y_{\ell-1}, y, x_j, \ldots, x_1\rangle) \in \mathrm{SAT}.$$

Since $\langle x_1, \ldots, x_j\rangle$ is a hard sequence for length m, by Lemma 4.1

$$\langle y_1, \ldots, y_{\ell-1}, y\rangle \in \mathrm{L}_{\mathrm{BH}(\ell)} \iff \pi_{(1,\ell)} \circ h(\langle y_1, \ldots, y_{\ell-1}, y, x_j, \ldots, x_1\rangle) \in \mathrm{L}_{\text{co-BH}(\ell)}.$$

If ℓ is even, then by the definitions of $\mathrm{L}_{\mathrm{BH}(\ell)}$ and $\mathrm{L}_{\text{co-BH}(\ell)}$,

(4)

$$\begin{array}{lcl} \langle y_1, \ldots, y_{\ell-1}\rangle \in \mathrm{L}_{\mathrm{BH}(\ell-1)} & & \pi_{(1,\ell-1)} \circ h(\langle y_1, \ldots, y_{\ell-1}, y, x_j, \ldots, x_1\rangle) \in \mathrm{L}_{\text{co-BH}(\ell-1)} \\ \quad \text{and } \ y \in \overline{\mathrm{SAT}} & \iff & \text{or } \ \pi_\ell \circ h(\langle y_1, \ldots, y_{\ell-1}, y, x_j, \ldots, x_1\rangle) \in \mathrm{SAT}. \end{array}$$

If ℓ is odd, then by the definitions of $\mathrm{L}_{\mathrm{BH}(\ell)}$ and $\mathrm{L}_{\text{co-BH}(\ell)}$,

(5)

$$\begin{array}{lcl} \langle y_1, \ldots, y_{\ell-1}\rangle \in \mathrm{L}_{\mathrm{BH}(\ell-1)} & & \pi_{(1,\ell-1)} \circ h(\langle y_1, \ldots, y_{\ell-1}, y, x_j, \ldots, x_1\rangle) \in \mathrm{L}_{\text{co-BH}(\ell-1)} \\ \quad \text{or } \ y \in \mathrm{SAT} & \iff & \text{and } \ \pi_\ell \circ h(\langle y_1, \ldots, y_{\ell-1}, y, x_j, \ldots, x_1\rangle) \in \overline{\mathrm{SAT}} \end{array}$$

or (by negating both sides of the iff)

(6)

$$\begin{array}{lcl} \langle y_1, \ldots, y_{\ell-1}\rangle \notin \mathrm{L}_{\mathrm{BH}(\ell-1)} & & \pi_{(1,\ell-1)} \circ h(\langle y_1, \ldots, y_{\ell-1}, y, x_j, \ldots, x_1\rangle) \notin \mathrm{L}_{\text{co-BH}(\ell-1)} \\ \quad \text{and } \ y \in \overline{\mathrm{SAT}} & \iff & \text{or } \ \pi_\ell \circ h(\langle y_1, \ldots, y_{\ell-1}, y, x_j, \ldots, x_1\rangle) \in \mathrm{SAT}. \end{array}$$

In either case, we already know by hypothesis that

$$\pi_\ell \circ h(\langle y_1, \ldots, y_{\ell-1}, y, x_j, \ldots, x_1\rangle) \in \mathrm{SAT},$$

so the right sides of the iff in equations (4) and (6) are satisfied. Therefore, the left sides of equations (4) and (6) must also be satisfied, and we have $y \in \overline{\mathrm{SAT}}$. □

Lemma 4.2 essentially states that a maximal hard sequence produces a way to witness that a formula is unsatisfiable. Hence, given a maximal hard sequence, an NP machine can guess these witnesses and verify that formulas up to a certain length are unsatisfiable. But if an NP machine can verify that formulas are unsatisfiable, it can simulate an $\mathrm{NP}^{\mathrm{NP}}$ computation by guessing the answer to each NP query and verifying that its answer is correct. We use this idea to prove Lemma 4.3, which states that given a maximal hard sequence, an NP machine can recognize an initial segment of L_{u_2}, the canonical complete language for Σ_2^{P}.

LEMMA 4.3. *Suppose h is a $\leq_{\mathrm{m}}^{\mathrm{P}}$-reduction from $\mathrm{L}_{\mathrm{BH}(k)}$ to $\mathrm{L}_{\text{co-BH}(k)}$. Then there exist an* NP *machine N_{σ_2} and a polynomial p_{σ_2} such that if $m \geq p_{\sigma_2}(|w|)$ and $\langle 1^m, x_1, \ldots, x_j\rangle$ is a maximal hard sequence w.r.t. h, then*

$$w \in L_{u_2} \iff N_{\sigma_2}(w, \langle 1^m, x_1, \ldots, x_j\rangle) \ \textit{accepts}.$$

Proof. Let $L_{u_2} = L(N_{u_2}^{\mathrm{SAT}})$. Define $p_{\sigma_2}(n)$ to be the upper bound on the running time of N_{u_2} on inputs of length n. Obviously, $N_{u_2}(w)$ queries only strings of length $\leq m$, since $m \geq p_{\sigma_2}(|w|)$. On input $(w, \langle 1^m, x_1, \ldots, x_j\rangle)$, N_{σ_2} does the following:

1. Simulate N_{u_2} step by step until N_{u_2} makes an oracle query to SAT.
2. When N_{u_2} queries "$y \in \text{SAT}$?", branch into two computations. One guesses that $y \in \text{SAT}$; the other guesses that $y \in \overline{\text{SAT}}$.
3. The branch that guesses $y \in \text{SAT}$ will guess a satisfying assignment for y. If none are found, all computations along this branch terminate. If a satisfying assignment is found, then the guess that $y \in \text{SAT}$ is correct and the simulation continues.
4. The branch that guesses $y \in \overline{\text{SAT}}$ will use the maximal hard sequence to find a witness for $y \in \overline{\text{SAT}}$, i.e., it guesses $\ell - 1$ strings $y_1, \ldots, y_{\ell-1} \in \Sigma^{\leq m}$, computes

$$F = \pi_\ell \circ h(\langle y_1, \ldots, y_{\ell-1}, y, x_j, \ldots, x_1 \rangle),$$

and guesses a satisfying assignment for F. If none are found, all computations along this branch terminate. Otherwise, the guess that $y \in \overline{\text{SAT}}$ is correct and the simulation continues.

By Lemma 4.2, if $\langle 1^m, x_1, \ldots, x_j \rangle$ is a maximal hard sequence, then for each query y, $y \in \overline{\text{SAT}}$ iff some computation in step 4 finds a satisfiable F. So, in the simulation of the oracle query, all the computations along one branch will terminate and some computations in the other branch will continue. Thus the simulation continues iff the guesses for the oracle answers (either $y \in \text{SAT}$ or $y \in \overline{\text{SAT}}$) are verified, and hence

$$w \in L(N_{u_2}^{\text{SAT}}) \iff N_{\sigma_2}(w, \langle 1^m, x_1, \ldots, x_j \rangle) \text{ accepts.} \qquad \square$$

Taking the spirit of Lemma 4.3 one step further, we show that with the help of a maximal hard sequence, an NP^{NP} machine can simulate a Σ_3^{P} machine. In addition, an NP^{NP} machine can guess and verify hard sequences, so it does not need to be given a maximal hard sequence; all it really needs is the *maximum order.* Therefore, there exists an NP^{NP} machine which, given the maximum order of hard sequences for a length, can recognize initial segments of L_{u_3}, the complete language for Σ_3^{P}.

LEMMA 4.4. *Suppose h is a $\leq_{\text{m}}^{\text{P}}$-reduction from $\text{L}_{\text{BH}(k)}$ to $\text{L}_{\text{co-BH}(k)}$. There exist an NP^{SAT} machine N_{σ_3} and a polynomial p_{σ_3} such that for any $m \geq p_{\sigma_3}(|w|)$, if j is the maximum order for length m w.r.t. h, then*

$$w \in L_{u_3} \iff N_{\sigma_3}^{\text{SAT}}(w, j, 1^m) \text{ accepts.}$$

Furthermore, if j is greater than the maximum order for length m w.r.t. h,

$$\forall w \;\; N_{\sigma_3}^{\text{SAT}}(w, j, 1^m) \text{ rejects.}$$

Proof. Let $L_{u_3} = L(N_{u_3}^{L_{u_2}})$, where $L_{u_2} = L(N_{u_2}^{\text{SAT}})$ is the canonical complete language for Σ_2^{P}. Let $r(n)$ be a polynomial upper bound on the running time of N_{u_3} on inputs of length n. Clearly, $N_{u_3}(w)$ will query only strings of length $\leq r(n)$, where $n = |w|$. Apply Lemma 4.3 to obtain N_{σ_2} and the polynomial p_{σ_2}. Let $p_{\sigma_3}(n) \overset{\text{def}}{=} p_{\sigma_2}(r(n))$.

The critical observation to make here is that the set of hard sequences is in co-NP. (This is obvious from the definition of hard sequences.) So, given j, the maximum order for length $m \geq p_{\sigma_3}(n)$, an NP^{NP} machine can guess j strings $x_1, \ldots, x_j \in \Sigma^{\leq m}$ and ask the NP oracle if $\langle 1^m, x_1, \ldots, x_j \rangle$ forms a hard sequence. If $\langle 1^m, x_1, \ldots, x_j \rangle$ does form a hard sequence, then it must also be a maximal sequence since it is of maximum order. Now, $N_{\sigma_3}^{\text{SAT}}(w, j, 1^m)$ can simulate $N_{u_3}^{L_{u_2}}(w)$ step by step, and when

N_{u_3} queries "$y \in L_{u_2}$?", N_{σ_3} will ask "$(y, \langle 1^m, x_1, \ldots, x_j\rangle) \in L(N_{\sigma_2})$." By Lemma 4.3 the two queries will return with the same answers, so

$$w \in L(N_{u_3}^{L_{u_2}}) \iff N_{\sigma_3}^{\mathrm{SAT}}(w, j, 1^m) \text{ accepts.}$$

Note that when N_{σ_3} guesses the hard sequence $\langle 1^m, x_1, \ldots, x_j\rangle$, several computation paths of the NP machine may survive because there may be many hard sequences of maximum order. However, uniqueness is not important here because any maximal hard sequence will work for N_{σ_2}. So, all the computation branches that manage to guess a hard sequence of maximum order will have the same acceptance behavior. Furthermore, if j is greater than the maximum order, then none of the computation paths survive because there are no hard sequences of order j for length m. Thus, in this case, $N_{\sigma_3}(w, j, 1^m)$ will reject. □

We have shown that maximal hard sequences and maximum orders expand the computational power of nondeterministic machines. We define the set T to be the set of strings encoding the orders of hard sequences for each length.

DEFINITION. *Suppose h is a $\leq_{\mathrm{m}}^{\mathrm{P}}$-reduction from $\mathrm{L}_{\mathrm{BH}(k)}$ to $\mathrm{L}_{\text{co-BH}(k)}$. We define an associated set T by*

$$T \stackrel{\text{def}}{=} \{(1^m, j) \mid \exists x_1, \ldots, x_j \in \Sigma^{\leq m}, \text{s.t. } \langle 1^m, x_1, \ldots, x_j\rangle \textit{ is a hard sequence.}\}$$

Note that since the set of hard sequences is in co-NP, T itself is in $\mathrm{NP}^{\mathrm{NP}}$. This gives us the following lemma.

LEMMA 4.5. *Suppose h is a $\leq_{\mathrm{m}}^{\mathrm{P}}$-reduction from $\mathrm{L}_{\mathrm{BH}(k)}$ to $\mathrm{L}_{\text{co-BH}(k)}$. Then the set T defined above is in* $\mathrm{NP}^{\mathrm{NP}}$. □

Since $T \in \mathrm{NP}^{\mathrm{NP}}$, a $\mathrm{P}^{\mathrm{NP}^{\mathrm{NP}}}$ machine can compute the order of the maximum hard sequence for length m with $k-1$ queries. The $\mathrm{P}^{\mathrm{NP}^{\mathrm{NP}}}$ machine can then pass this number to the $\mathrm{NP}^{\mathrm{NP}}$ machine $N_{\sigma_3}^{\mathrm{SAT}}$ of Lemma 4.4 to recognize L_{u_3}, the complete language for Σ_3^{P}. Therefore, if the BH collapses to its kth level, Lemmas 4.4 and 4.5 imply that the PH collapses to $\mathrm{P}^{\mathrm{NP}^{\mathrm{NP}}[k]}$. This collapse of the PH implied by the collapse of the BH is lower than previously known.

THEOREM 4.6. *Suppose h is a $\leq_{\mathrm{m}}^{\mathrm{P}}$-reduction from $\mathrm{L}_{\mathrm{BH}(k)}$ to $\mathrm{L}_{\text{co-BH}(k)}$. Then there exists a $\mathrm{P}^{\mathrm{NP}^{\mathrm{NP}}}$ machine which accepts L_{u_3} with only k queries to the $\mathrm{NP}^{\mathrm{NP}}$ oracle. That is, the PH collapses to $\mathrm{P}^{(\mathrm{NP}^{\mathrm{NP}})[k]}$.*

Proof. By Lemma 4.4, there exists N_{σ_3} and p_{σ_3} such that if j is the maximum order for length m and $m \geq p_{\sigma_3}(|w|)$, then

$$w \in L_{u_3} \iff N_{\sigma_3}^{\mathrm{SAT}}(w, j, 1^m) \text{ accepts.}$$

Using the fact that T is in $\mathrm{NP}^{\mathrm{NP}}$ (Lemma 4.5), a $\mathrm{P}^{\mathrm{NP}^{\mathrm{NP}}}$ machine can determine if $(1^m, \ell)$ is in T by asking the oracle. Doing this for all values of ℓ between 1 and $k-1$, it can determine the maximum ℓ such that $(1^m, \ell)$ is in T. This maximum ℓ—call it j—is, of course, the maximum order for length m. Then with one final query, the $\mathrm{P}^{\mathrm{NP}^{\mathrm{NP}}}$ machine asks if

$$N_{\sigma_3}^{\mathrm{SAT}}(w, j, 1^m) \text{ accepts.}$$

If the oracle answers "yes," the machine accepts. Otherwise, it rejects. □

Note that we could make Theorem 4.6 stronger by using binary search instead of linear search to find the maximum order. However, we will push the collapse even further in Theorem 4.9, so our inquiry will follow a new direction.

The following lemma states that an NP machine can recognize if there is a hard sequence of order j for length m if it is given a maximal hard sequence for a longer length.

LEMMA 4.7. *Suppose h is a $\leq_{\rm m}^{\rm P}$-reduction from $\rm L_{BH(k)}$ to $\rm L_{co\text{-}BH(k)}$. There exist an* NP *machine N_t and a polynomial p_t such that if $\langle 1^{m_2}, \vec{x}\,\rangle$ is a maximal hard sequence w.r.t. h and $m_2 \geq p_t(m_1 + k)$, then*

$$(1^{m_1}, j_1) \in T \iff N_t((1^{m_1}, j_1), \langle 1^{m_2}, \vec{x}\,\rangle) \text{ accepts.}$$

Proof. Use Lemmas 4.3 and 4.5. □

In Lemma 4.4 we showed that, with the help of the maximum order, an $\rm NP^{NP}$ machine can recognize a complete language for $\Sigma_3^{\rm P}$. In the next lemma we show that with the help of the maximum order, an $\rm NP^{NP}$ machine can also recognize a complete language for $\Pi_3^{\rm P}$ (i.e., recognize when a string is *not* in L_{u_3}).

LEMMA 4.8. *Suppose h is a $\leq_{\rm m}^{\rm P}$-reduction from $\rm L_{BH(k)}$ to $\rm L_{co\text{-}BH(k)}$. Let L_{u_3} be the canonical complete language for $\Sigma_3^{\rm P}$. There exist an $\rm NP^{SAT}$ machine N_{π_3} and a polynomial p_{π_3} such that for any $m \geq p_{\pi_3}(|w|)$, if j is the maximum order for length m w.r.t. h, then*

$$w \in \overline{L_{u_3}} \iff N_{\pi_3}^{\rm SAT}(w, j, 1^m) \text{ accepts.}$$

Furthermore, if j is greater than the maximum order for length m w.r.t. h,

$$\forall w \quad N_{\pi_3}^{\rm SAT}(w, j, 1^m) \text{ rejects.}$$

Proof. Let $L_{u_3} = L(N_{u_3}^{L_{u_2}})$ where L_{u_2} is the canonical complete language for $\Sigma_2^{\rm P}$. By Lemma 4.4, there exist N_{σ_3} and p_{σ_3} such that if j_1 is the maximum order for length m_1 and $m_1 \geq p_{\sigma_3}(|w|)$, then

$$w \in L_{u_3} \iff N_{\sigma_3}^{\rm SAT}(w, j_1, 1^{m_1}) \text{ accepts.}$$

The language accepted by $N_{\sigma_3}^{\rm SAT}$ is in $\Sigma_2^{\rm P}$, so we can reduce it to L_{u_2} via some polynomial-time function g. Let $r(n)$ be an upper bound on the running time of g. Using Lemma 4.3, we see that there exist N_{σ_2} and p_{σ_2} such that if $\langle 1^{m_2}, \vec{y}\,\rangle$ is a maximal hard sequence and $m_2 \geq p_{\sigma_2}(r(|w| + k + m_1))$, then

$$N_{\sigma_3}^{\rm SAT}(w, j_1, 1^{m_1}) \text{ accepts} \iff N_{\sigma_2}(g(w, j_1, 1^{m_1}), \langle 1^{m_2}, \vec{y}\,\rangle) \text{ accepts.}$$

Let N_s be the NP machine that runs the reduction g and then simulates N_{σ_2}, i.e.,

$$N_s(w, j_1, 1^{m_1}, \langle 1^{m_2}, \vec{y}\,\rangle) \text{ accepts} \iff N_{\sigma_2}(g(w, j_1, 1^{m_1}), \langle 1^{m_2}, \vec{y}\,\rangle) \text{ accepts.}$$

Let $p_s \stackrel{\rm def}{=} p_{\sigma_2} \circ r$. Now, if $m_1 \geq p_{\sigma_3}(|w|)$, $m_2 \geq p_s(|w| + k + m_1)$, j_1 is the maximum order for length m_1 and $\langle 1^{m_2}, \vec{y}\,\rangle$ is a maximal hard sequence, then

$$(7)\ w \in L_{u_3} \iff N_{\sigma_3}^{\rm SAT}(w, j_1, 1^{m_1}) \text{ accepts} \iff N_s(w, j_1, 1^{m_1}, \langle 1^{m_2}, \vec{y}\,\rangle) \text{ accepts.}$$

We are trying to prove that there exists a machine $N_{\pi_3}^{\rm SAT}$ that accepts $(w, j, 1^m)$ if $w \notin L_{u_3}$ when m is big enough in relation to $|w|$ and j is the maximum order of the hard sequences for length m. The $N_{\pi_3}^{\rm SAT}$ that we have in mind will map

$$(w, j, 1^m) \longrightarrow (w, j_1, 1^{m_1}, \langle 1^{m_2}, \vec{y}\,\rangle)$$

and accept iff $N_s(w, j_1, 1^{m_1}, \langle 1^{m_2}, \vec{y}\rangle)$ rejects (iff $w \notin L_{u_3}$). $N_{\pi_3}^{\mathrm{SAT}}$ can tell if $N_s(w, j_1, 1^{m_1}, \langle 1^{m_2}, \vec{y}\rangle)$ rejects with one query to SAT.

The difficulty in mapping $(w, j, 1^m) \longrightarrow (w, j_1, 1^{m_1}, \langle 1^{m_2}, \vec{y}\rangle)$ lies in the fact that $N_{\pi_3}^{\mathrm{SAT}}$ is given j, the maximum order of hard sequences for one length m, and it must compute the maximum orders of two other lengths, m_1 and m_2. We will define p_{π_3} so that if $m \geq p_{\pi_3}(|w|)$, then m will be bigger enough than both m_1 and m_2 so that we can apply Lemma 4.7 to compute j_1 and j_2.

Let $p_{\pi_3}(n) \overset{\mathrm{def}}{=} p_t(p_s(n + k + p_{\sigma_3}(n)) + k)$, i.e., $p_{\pi_3}(n) = p_t(m_2 + k)$ where $m_2 = p_s(n+k+m_1)$ and $m_1 = p_{\sigma_3}(n)$ (recall p_t is the polynomial bound from Lemma 4.7).

$N_{\pi_3}^{\mathrm{SAT}}(w, j, 1^m)$ will do the following. (We will annotate the program with a description of what $N_{\pi_3}^{\mathrm{SAT}}(w, j, 1^m)$ accomplishes when j is the maximum order.)

1. Reject if $m < p_{\pi_3}(|w|)$.
2. Guess j strings $x_1, \ldots, x_j \in \Sigma^{\leq m}$ and confirm that $\langle 1^m, x_1, \ldots, x_j\rangle = \langle 1^m, \vec{x}\rangle$ is a hard sequence by asking the SAT oracle. (Recall that checking if a given tuple forms a hard sequence is a co-NP question.) If j is the maximum order and $\langle 1^m, \vec{x}\rangle$ is a hard sequence, then $\langle 1^m, \vec{x}\rangle$ is a maximal hard sequence, too.
3. Let $n = |w|$. Compute $m_1 = p_{\sigma_3}(n)$ and $m_2 = p_s(n + k + m_1)$.
4. For $\ell = 0$ to $k - 1$, ask SAT if $N_t((1^{m_1}, \ell), \langle 1^m, \vec{x}\rangle)$ accepts. Let j_1 be the maximum ℓ where $N_t((1^{m_1}, \ell), \langle 1^m, \vec{x}\rangle)$ does accept. Note that $m = p_t(m_2 + k) \geq p_t(m_1 + k)$, so j_1 is the maximum order for length m_1 (by Lemma 4.7) if j is the maximum order for length m.
5. For $\ell = 0$ to $k - 1$, ask SAT if $N_t((1^{m_2}, \ell), \langle 1^m, \vec{x}\rangle)$ accepts. Let j_2 be the maximum ℓ where $N_t((1^{m_2}, \ell), \langle 1^m, \vec{x}\rangle)$ does accept. As in step 4, j_2 is the maximum order for length m_2 if j is the maximum order for length m.
6. Guess j_2 strings $y_1, \ldots, y_{j_2} \in \Sigma^{\leq m_2}$ and confirm that $\langle 1^{m_2}, y_1, \ldots, y_{j_2}\rangle = \langle 1^{m_2}, \vec{y}\rangle$ is a hard sequence (with one query to SAT). Note that if $\langle 1^{m_2}, \vec{y}\rangle$ is a hard sequence and j_2 is the maximum order, then $\langle 1^{m_2}, \vec{y}\rangle$ is also a maximal hard sequence.
7. Ask SAT if $N_s(w, j_1, 1^{m_1}, \langle 1^{m_2}, \vec{y}\rangle)$ accepts. If SAT returns "no," then $N_{\pi_3}^{\mathrm{SAT}}$ accepts. Note that by the preceding discussion, if j is the maximum order for length m, then j_1 is the maximum order for length m_1 and $\langle 1^{m_2}, \vec{y}\rangle$ is a maximal hard sequence. Also, $m_1 = p_{\sigma_3}(|w|)$ and $m_2 = p_s(|w| + k + m_1)$, so by equation (7)

$$w \in L_{u_3} \iff N_s(w, j_1, 1^{m_1}, \langle 1^{m_2}, \vec{y}\rangle) \text{ accepts.}$$

Now, we argue that if j is the maximum order for length m and $m \geq p_{\pi_3}(|w|)$, then

$$w \in \overline{L_{u_3}} \iff N_{\pi_3}^{\mathrm{SAT}}(w, j, 1^m) \text{ accepts.}$$

First of all, $N_{\pi_3}^{\mathrm{SAT}}$ accepts in step 7 only. So, if $w \in L_{u_3}$, all computation paths of $N_{\pi_3}^{\mathrm{SAT}}$ reject—even those that reach step 7, because SAT would answer "yes" in step 7. On the other hand, if $w \in \overline{L_{u_3}}$, then some computation path will reach step 7, get "no" from the SAT oracle, and accept.

Finally, we note that if j is greater than the maximum order for length m, then no computation path will survive step 2. Thus, in this case $N_{\pi_3}^{\mathrm{SAT}}(w, j, 1^m)$ rejects. □

Now we are ready to prove our main theorem. This theorem demonstrates a close linkage between the collapse of the BH and the PH.

THEOREM 4.9. *Suppose h is a $\leq_{\mathrm{m}}^{\mathrm{P}}$-reduction from $\mathrm{L}_{\mathrm{BH}(k)}$ to $\mathrm{L}_{\text{co-BH}(k)}$. Let L_{u_3} be the canonical complete language for Σ_3^{P}. Then there exist languages $B_1, \ldots, B_k \in \mathrm{NP}^{\mathrm{NP}}$ such that*

$$L_{u_3} = B_1 - (B_2 - (B_3 - (\cdots - B_k))).$$

That is, $\Sigma_3^{\mathrm{P}} \subseteq \mathrm{BH}_3(k)$, and therefore $\mathrm{PH} \subseteq \mathrm{BH}_3(k)$.

Proof. First, recall that in Lemmas 4.4 and 4.8 it was shown that $N_{\sigma_3}^{\mathrm{SAT}}$ and $N_{\pi_3}^{\mathrm{SAT}}$ accepted L_{u_3} and $\overline{L_{u_3}}$ (respectively) with the help of the maximum order for a large enough length (and they reject if the number given for the maximum order is too large). Let w be any string. Let $m = \max(p_{\sigma_3}(|w|), p_{\pi_3}(|w|))$; then m is large enough so that if j is the maximum order for length m,

$$N_{\sigma_3}^{\mathrm{SAT}}(w, j, 1^m) \text{ accepts } \iff w \in L_{u_3},$$

$$N_{\pi_3}^{\mathrm{SAT}}(w, j, 1^m) \text{ accepts } \iff w \notin L_{u_3}.$$

We will define the $\mathrm{NP}^{\mathrm{NP}}$ languages $B_1, \ldots, B_k$ to be the strings accepted by $\mathrm{NP}^{\mathrm{NP}}$ machines that try to guess j, the maximum order for length m, and then run N_{σ_3} and N_{π_3}. These $\mathrm{NP}^{\mathrm{NP}}$ machines cannot verify when they have guessed the true maximum order; instead, they will base their acceptance behavior on whether they can determine that an earlier machine in the sequence may have been fooled by an incorrect guess for j. This successive approximation scheme converges to the language L_{u_3} within k steps.

DEFINITION. *For $1 \leq \ell \leq k$, the language B_ℓ is the set of strings w with the property that there exist $j_1, \ldots, j_\ell$ such that*

1. $0 \leq j_1 < j_2 < \cdots < j_\ell \leq k-1$,
2. *for all odd d, $1 \leq d \leq \ell$, $N_{\sigma_3}^{\mathrm{SAT}}(w, j_d, 1^m)$ accepts,*
3. *for all even d, $1 \leq d \leq \ell$, $N_{\pi_3}^{\mathrm{SAT}}(w, j_d, 1^m)$ accepts.*

Clearly, B_ℓ is in $\mathrm{NP}^{\mathrm{NP}}$ since an $\mathrm{NP}^{\mathrm{NP}}$ machine can guess $j_1, \ldots, j_\ell$, verify the first property, and then simulate $N_{\sigma_3}^{\mathrm{SAT}}$ and $N_{\pi_3}^{\mathrm{SAT}}$ for the different values of j_d. Also, observe that the B_ℓ's form a nested sequence

$$B_k \subseteq B_{k-1} \subseteq \cdots \subseteq B_2 \subseteq B_1.$$

Finally, note that if $r = \max\{\ell \mid w \in B_\ell\}$, then

$$w \in B_1 - (B_2 - (B_3 - (\cdots - B_k))) \iff r \text{ is odd}.$$

Example. Here we give an example which demonstrates that

$$w \in L_{u_3} \iff r = \max\{\ell \mid w \in B_\ell\} \text{ is odd}.$$

Let $k = 8$, the maximum order for length m be 5, and $N_{\sigma_3}^{\mathrm{SAT}}$ and $N_{\pi_3}^{\mathrm{SAT}}$ behave as shown in Table 1 below for the different values of s plugged in as the guess for the maximum order.

Note that since the maximum order is 5, both $N_{\sigma_3}^{\mathrm{SAT}}$ and $N_{\pi_3}^{\mathrm{SAT}}$ reject for $s = 6, 7$. Also, one of $N_{\sigma_3}^{\mathrm{SAT}}(w, 5, 1^m)$ and $N_{\pi_3}^{\mathrm{SAT}}(w, 5, 1^m)$ must accept and the other reject. For smaller s, both may accept or reject, since their behavior is unpredictable. Finally, $w \in L_{u_3}$, so we want show that r is odd.

To determine if $w \in B_\ell$, look for an alternating (between the top and bottom row) sequence of accepts starting from the top row moving left to right. If there is such a sequence of ℓ acceptances, then $w \in B_\ell$. In this example, $r = 3$.

TABLE 1

$s =$	0	1	2	3	4	5	6	7
$N_{\sigma_3}^{\mathrm{SAT}}(w, s, 1^m)$	rej	acc	acc	rej	rej	acc	rej	rej
$N_{\pi_3}^{\mathrm{SAT}}(w, s, 1^m)$	acc	rej	acc	rej	acc	rej	rej	rej

- $w \in B_1$, because j_1 can be 1, 2, or 5.
- $w \in B_2$, because both $N_{\sigma_3}^{\mathrm{SAT}}(w, 1, 1^m)$ and $N_{\pi_3}^{\mathrm{SAT}}(w, 4, 1^m)$ accept.
- $w \in B_3$ with $j_1 = 1, j_2 = 2, j_3 = 5$.
- $w \notin B_4, B_5, \ldots, B_8$, because there is no alternating sequence longer than 3. The sequence $j_1 = 0, j_2 = 1, j_3 = 2, j_4 = 5$ does not count because the sequence must start from the top row.

CLAIM 4.10. *If $w \in L_{u_3}$, then $r = \max\{\ell \mid w \in B_\ell\}$ is odd.*

Proof. Let j be the maximum order for length m. Now suppose r is even and $w \in B_r$. Then, there exist $j_1, \ldots, j_r$ so that properties 1–3 in the definition above hold. Therefore,

$$N_{\pi_3}^{\mathrm{SAT}}(w, j_r, 1^m) \text{ accepts}$$

(since r is even and $w \in B_r$). Since $w \in L_{u_3}$, for the true maximum order j,

$$N_{\pi_3}^{\mathrm{SAT}}(w, j, 1^m) \text{ rejects.}$$

Therefore $j_r \neq j$. Observe that j_r cannot be greater than j either since for all $s > j$,

$$N_{\pi_3}^{\mathrm{SAT}}(w, s, 1^m) \text{ rejects.}$$

Hence $j_r < j$.

Since we are given that $w \in L_{u_3}$, we know that $N_{\sigma_3}^{\mathrm{SAT}}(w, j, 1^m)$ must accept (Lemma 4.4). Now consider the sequence $j_1, \ldots, j_{r+1}$, where $j_{r+1} = j$. $N_{\sigma_3}^{\mathrm{SAT}}(w, j_{r+1}, 1^m)$ accepts and $r + 1$ is odd, which implies that $j_1, \ldots, j_{r+1}$ satisfies conditions 1–3, and therefore $w \in B_{r+1}$. Thus if r is even, $r \neq \max\{\ell \mid w \in B_\ell\}$. Therefore, r must be odd.

CLAIM 4.11. *If $w \notin L_{u_3}$ then $r = \max\{\ell \mid w \in B_\ell\}$ is even.*

Proof. This is similar to the proof of Claim 4.10.

Combining Claims 4.10 and 4.11 with the observation that if $r = \max\{\ell \mid w \in B_\ell\}$, then

$$w \in B_1 - (B_2 - (B_3 - (\cdots - B_k))) \iff r \text{ is odd},$$

we have

$$w \in L_{u_3} \iff w \in B_1 - (B_2 - (B_3 - (\cdots - B_k))). \qquad \square$$

Theorem 4.9 also shows some unexpected connections between the Boolean and query hierarchies within Δ_2^{P} and Δ_3^{P}.

COROLLARY 4.12. $\mathrm{BH}(k) = \text{co-BH}(k) \Longrightarrow \mathrm{BH}_3(k) = \text{co-BH}_3(k)$.

COROLLARY 4.13. $\mathrm{P}^{\mathrm{NP}\|[k]} = \mathrm{P}^{\mathrm{NP}\|[k+1]} \Longrightarrow \mathrm{P}^{\mathrm{NP}^{\mathrm{NP}}\|[k+1]} = \mathrm{P}^{\mathrm{NP}^{\mathrm{NP}}\|[k+2]}$.

Proof. We use the fact that the Boolean and query hierarchies are intertwined:

$$\mathrm{P}^{\mathrm{NP}\|[k]} \subseteq \mathrm{BH}(k+1) \subseteq \mathrm{P}^{\mathrm{NP}\|[k+1]} \quad \text{and} \quad \mathrm{BH}_3(k+1) \subseteq \mathrm{P}^{\mathrm{NP}^{\mathrm{NP}}\|[k+1]}.$$

If $\mathrm{P}^{\mathrm{NP}\|[k]} = \mathrm{P}^{\mathrm{NP}\|[k+1]}$, then $\mathrm{BH}(k+1)$ is closed under complementation. Moreover, Corollary 4.12 implies that $\mathrm{BH}_3(k+1)$ is closed under complementation as well. Thus, the Boolean and query hierarchies in Δ_3^{P} collapse to $\mathrm{BH}_3(k+1)$. These simple containments give a result that is off by one. However, we believe that with more effort we should be able to show that $\mathrm{P}^{\mathrm{NP}^{\mathrm{NP}}\|[k]} = \mathrm{P}^{\mathrm{NP}^{\mathrm{NP}}\|[k+1]}$. □

5. Conclusion. We have demonstrated a closer connection between the BH and the PH—a deeper collapse of the BH implies a deeper collapse of the PH. We would like to think that this relationship is a consequence of some underlying structure connecting $\mathrm{BH}(k)$ and $\mathrm{BH}_3(k)$. However, attempts to simplify the proof along these lines have failed. Is there some straightforward argument which would show that $\mathrm{BH}(k) = \text{co-}\mathrm{BH}(k)$ implies $\mathrm{BH}_3(k)$ is closed under complementation? Could such an argument be extended to show that $\Sigma_3^{\mathrm{P}} \subseteq \mathrm{BH}_3(k)$? Finally, we ask, is this collapse optimal (for $k \geq 2$) or can it be shown that $\mathrm{BH}(k) = \text{co-}\mathrm{BH}(k)$ implies $\mathrm{PH} \subseteq \mathrm{BH}(k)$?

Acknowledgments. We are grateful to Juris Hartmanis for his support and guidance. We would also like to thank Stephen Mahaney for starting us on this endeavor and Georges Lauri for reading a draft of this paper.

REFERENCES

[1] T. Gundermann, J. Hartmanis, L. Hemachandra, V. Sewelson, K. Wagner, and G. Wechsung, *Boolean hierarchy I: Structural properties*, SIAM J. Comput., 17 (1988), pp. 1232–1252.

[2] J. Cai and L. A. Hemachandra, *The Boolean hierarchy: Hardware over NP*, in Structure in Complexity Theory, Lecture Notes in Comput. Sci., 223 (1986), pp. 105–124.

[3] J. Kadin, *The polynomial time hierarchy collapses if the Boolean hierarchy collapses*, SIAM J. Comput., 17 (1988), pp. 1263–1282.

[4] ———, *ERRATUM: The polynomial time hierarchy collapses if the Boolean hierarchy collapses*, SIAM J. Comput., 20 (1991), p. 404.

[5] ———, *Restricted Turing reducibilities and the structure of the polynomial time hierarchy*, Ph.D. thesis, Cornell University, 1988.

[6] S. Mahaney, 1989, private communication.

[7] K. Wagner and G. Wechsung, *On the Boolean closure of NP*, in Proc. 1985 International Conference on Fundamentals of Computation Theory, Lecture Notes in Comput. Sci., 199 (1985), pp. 485–493.

[8] C. Yap, *Some consequences of non-uniform conditions on uniform classes*, Theoret. Comput. Sci., 26 (1983), pp. 287–300.

SIAM J. COMPUT.
Vol. 25, No. 2, pp. 355–368, April 1996

LOW-DEGREE SPANNING TREES OF SMALL WEIGHT*

SAMIR KHULLER[†], BALAJI RAGHAVACHARI[‡], AND NEAL YOUNG[§]

Abstract. Given n points in the plane, the degree-K spanning-tree problem asks for a spanning tree of minimum weight in which the degree of each vertex is at most K. This paper addresses the problem of computing low-weight degree-K spanning trees for $K > 2$. It is shown that for an arbitrary collection of n points in the plane, there exists a spanning tree of degree 3 whose weight is at most 1.5 times the weight of a minimum spanning tree. It is shown that there exists a spanning tree of degree 4 whose weight is at most 1.25 times the weight of a minimum spanning tree. These results solve open problems posed by Papadimitriou and Vazirani. Moreover, if a minimum spanning tree is given as part of the input, the trees can be computed in $O(n)$ time.

The results are generalized to points in higher dimensions. It is shown that for any $d \geq 3$, an arbitrary collection of points in $\Re^d$ contains a spanning tree of degree 3 whose weight is at most 5/3 times the weight of a minimum spanning tree. This is the first paper that achieves factors better than 2 for these problems.

Key words. algorithms, graphs, spanning trees, approximation algorithms, geometry

AMS subject classifications. 05C05, 05C10, 05C85, 65Y25, 68Q20, 68R10, 68U05, 90C27, 90C35

1. Introduction.

Given n points in the plane, how do we find a spanning tree of minimum weight among those in which each vertex has degree at most K? Here the weight of an edge between two points is defined to be the Euclidean distance between them. This problem is referred to as the *Euclidean degree-K spanning tree problem* and is a generalization of the Hamilton path problem, which is known to be NP-hard [10, 12]. When $K = 3$, it was shown to be NP-hard by Papadimitriou and Vazirani [15], who conjectured that it is NP-hard for $K = 4$ as well. When $K = 5$, the problem can be solved in polynomial time [14].

This paper addresses the problem of computing low-weight degree-K spanning trees for $K > 2$. In any metric space, it is known that there always exists a spanning tree of degree 2 whose cost is at most twice the cost of a minimum spanning tree (MST). This is shown by taking a Euler tour of an MST (in which each edge is taken twice) and producing a Hamilton tour by short-cutting the Euler tour. In the case of general metric spaces, it is easy to generate examples in which the ratio of a shortest Hamilton path to the weight of an MST is arbitrarily close to 2. But such examples do not translate to points in $\Re^d$. In view of this, Papadimitriou and Vazirani [15] posed the problem of obtaining factors better than 2 for the Euclidean degree-K spanning-tree problem. It should be noted that in the special case of $K = 2$, Christofides [3] gave a simple and elegant polynomial-time approximation algorithm with an approximation ratio of 1.5 for computing a traveling salesperson tour for points satisfying the triangle inequality (points in a metric space).

* Received by the editors February 3, 1994; accepted for publication (in revised form) August 31, 1994.

[†] Department of Computer Science and Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742 (samir@cs.umd.edu). The research of this author was supported by NSF Research Initiation Award CCR-9307462.

[‡] Department of Computer Science, University of Texas at Dallas, Richardson, TX 75083 (rbk@utdallas.edu). The research of this author was supported by NSF Research Initiation Award CCR-9409625.

[§] Department of Computer Science, Dartmouth College, Hanover, NH 03755-3510 (neal.young@dartmouth.edu). This research was done while the author was at Cornell University and at UMIACS and was supported in part by NSF grants CCR-8906949 and CCR-9111348.

1.1. Our contributions. In this paper, we show that for an arbitrary collection of n points in the plane, there exists a degree-3 spanning tree whose weight is at most 1.5 times the weight of an MST. We also show that there exists a degree-4 spanning tree whose weight is at most 1.25 times the weight of an MST. This solves an open problem posed by Papadimitriou and Vazirani [15].

Moreover, if an MST is given as part of the input, the trees can be computed in $O(n)$ time. Note that our bound of 1.5 for the degree-3 spanning-tree problem is an "absolute" guarantee (based on the weight of an MST) as opposed to a "relative" guarantee for the degree-2 spanning tree obtained by Christofides [3] (based on the weight of an optimal solution).

We also generalize our results to points in higher dimensions. We show that for any $d \geq 2$, an arbitrary collection of points in $\Re^d$ contains a degree-3 spanning tree whose weight is at most 5/3 times the weight of an MST. This is the first paper that achieves factors better than 2 for these problems.

1.2. Significance of our results. Many approximation algorithms make use of the triangle inequality to obtain approximate solutions to NP-hard problems. These algorithms typically involve a "short-cutting" step where the triangle inequality is used to bound the cost of the obtained solution. Examples include Christofides's heuristic for the traveling salesperson problem [3], biconnectivity augmentation [8], approximate weighted matching [11], prize-collecting traveling salesperson [2], and bounded-degree subgraphs which have low weight and small bottleneck cost [16].

A question of general interest is how to obtain improved approximation algorithms for such problems when the points come from a Euclidean, as opposed to arbitrary, metric space. This requires making use of more than just the triangle inequality. Surprisingly, for most problems, improved algorithms are not known. (A notable exception is the famous Euclidean Steiner tree problem [5, 6].) We use rudimentary geometric techniques to obtain an improved algorithm for the Euclidean degree-K spanning-tree problem.

The key to our method is to give short-cutting steps that are provably better than implied by the triangle inequality alone. Lemma 3.3, which bounds the perimeter of an arbitrary triangle in terms of distances to its vertices from any point, is typical of the techniques that we use to get better bounds.

1.3. Related work. Papadimitriou and Vazirani showed that any MST whose vertices have integer coordinates has maximum degree at most 5 [15]. Monma and Suri [14] showed that for *every* set of points in the plane, there exists a degree-5 MST.

Many recent works have given algorithms to find subgraphs of bounded degree that simultaneously satisfy other given constraints. A polynomial-time algorithm to find a spanning tree or a Steiner tree of a given subset of vertices in a graph with degree at most one more than minimum was given by Fürer and Raghavachari [9]. This was extended to weighted graphs by Fischer [7]. He showed how to find MSTs whose degree is within a constant multiplicative factor plus an additive $O(\log n)$ of the optimal degree. The degree bound is improved further in the case when the number of different edge weights is bounded by a constant. Ravi et al. [16] consider the problem of computing bounded-degree subgraphs satisfying given connectivity properties in a graph whose edge weights satisfy the triangle inequality. They give efficient algorithms for computing subgraphs which have low weight and small bottleneck cost. Salowe [18] and Das and Heffernan [4] consider the problem of computing bounded-degree graph spanners and provide algorithms for computing them. Robins and Salowe [17] study the maximum degrees of MSTs under various metrics.

2. Preliminaries. Let $V = \{v_1, \ldots, v_n\}$ be a set of n points in the plane. Let G be the complete graph induced by V, where the weight of an edge is the Euclidean distance between its endpoints. We use the terms points and vertices interchangeably. Let $\overline{uv}$ be the Euclidean distance between vertices u and v. Let $T_{\min}$ be an MST of the points in V. Let $w(T)$ denote the total weight of a spanning tree T. Let T_k denote a spanning tree in which every vertex has degree at most k. Let $\deg_T(v)$ be the degree of a vertex v in the tree T. Let ΔABC denote the triangle formed by points A, B, and C. Let $\angle ABC$ denote the angle formed at B between line segments AB and BC. Let $\overline{ABC}$ denote the perimeter of ΔABC; and more generally, let $\overline{v_1 v_2 \ldots v_k}$ denote the perimeter of the polygon formed by the line segments $v_i v_{i+1}$ for $1 \leq i \leq k$, where $v_{k+1} = v_1$.

In this paper, we prove the following: for an arbitrary set of points in $\Re^2$,

$$\exists T_3 : \quad w(T_3) \leq 1.5 \times w(T_{\min}), \tag{1}$$

$$\exists T_4 : \quad w(T_4) \leq 1.25 \times w(T_{\min}). \tag{2}$$

For an arbitrary set of points in $\Re^d$ $(d > 2)$,

$$\exists T_3 : \quad w(T_3) \leq \frac{5}{3} \times w(T_{\min}). \tag{3}$$

3. Points in the plane. We first consider the case of $\Re^2$—points in the plane. We first note some useful properties of MSTs in $\Re^d$.

PROPOSITION 3.1 ([15]). *Let AB and BC be two edges incident to a point B in an MST of a set of points in $\Re^d$. Then $\angle ABC$ is a largest angle in ΔABC.*

COROLLARY 3.2. *Let AB and BC be two edges incident to a point B in an MST of a set of points in $\Re^d$. Then*

- *$\angle ABC \geq 60°$,*
- *$\angle BAC, \angle BCA \leq 90°$.*

3.1. An upper bound on the perimeter of a triangle. We now prove an upper bound on the perimeter of an arbitrary triangle in terms of distances to its vertices from an arbitrary point. This lemma is useful in proving the performances of our algorithms. The lemma is also interesting in its own right, and we believe that it and the associated techniques will be useful in other geometrical problems.

LEMMA 3.3. *Let X, A, B, and C be points in $\Re^d$ with $\overline{XA} \leq \overline{XB}, \overline{XC}$. Then*

$$\overline{ABC} \leq (3\sqrt{3} - 4)\overline{XA} + 2(\overline{XB} + \overline{XC}). \tag{4}$$

Note that $3\sqrt{3} - 4 \approx 1.2$. Recall that $\overline{ABC}$ is the perimeter of the triangle and $\overline{XY}$ is the distance from X to Y.

Proof. Let B' and C' be points on XB and XC, respectively, such that $\overline{XA} = \overline{XB'} = \overline{XC'}$ (see Fig. 1). First we observe that the lemma is true if it is true for the points X, A, B', and C'. This follows because by the triangle inequality,

$$\overline{ABC} \leq \overline{AB'C'} + 2\overline{BB'} + 2\overline{CC'}.$$

By our assumption,

$$\overline{AB'C'} \leq (3\sqrt{3} - 4)\overline{XA} + 2(\overline{XB'} + \overline{XC'}).$$

Combining the two inequalities yields the desired result. Therefore, in the rest of the proof, we show that the lemma is true when the "arms" $\overline{XA}$, $\overline{XB'}$, and $\overline{XC'}$ are equal.

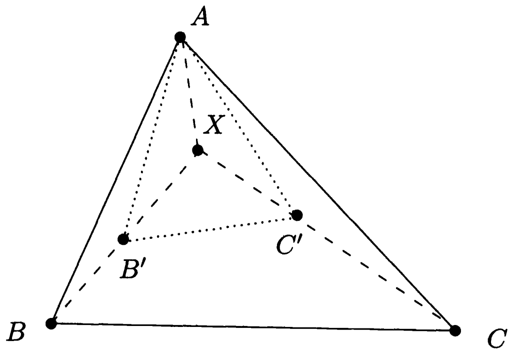


FIG. 1. *Shrinking to obtain canonical form.*

It is not very difficult to see that to maximize the perimeter of the triangle, X will be in the plane defined by A, B', and C', and thus X is at the center of a circle passing through A, B', and C'.

By scaling, it suffices to consider the case when the circle has unit radius. In this case, the right-hand side (r.h.s.) of (4) is exactly $3\sqrt{3}$. Thus, it suffices to show that the maximum perimeter achieved by any triangle whose vertices lie on a unit circle is $3\sqrt{3}$. This is easily proved [13]. □

Note that in an arbitrary metric space it is possible to have an (equilateral) triangle of perimeter six and a point X at distance one from each vertex.

3.2. Spanning trees of degree 3. We now assume that we are given a Euclidean MST T of degree at most 5. We show how to convert T into a tree of degree at most 3. The weight of the resulting tree is at most 1.5 times the weight of T.

High level description. The tree T is rooted at an arbitrary leaf vertex. Since T is a degree-5 tree, once it is rooted at a leaf, each vertex has at most four children. For each vertex v, the shortest path P_v starting at v and visiting every child of v is computed. The final tree T_3 consists of the union of the paths $\{P_v\}$. Figure 2 gives the above algorithm. In analyzing the algorithm, we think of each vertex v as replacing its edges from its children with the path P_v. The above technique of "short-cutting" the children of a vertex by "stringing" them together has been known before, especially in the context of computing degree-3 trees in metric spaces (see [16, 18]).

```
TREE-3($V,T$) — Find a degree-3 tree of $V$.
Root the MST $T$ at a leaf vertex $r$.
For each vertex $v \in V$ do
    Compute $P_v$, the shortest path starting at $v$ and visiting all the children of $v$.
Return $T_3$, the tree formed by the union of the paths $\{P_v\}$.
```

FIG. 2. *Algorithm to find a degree-3 tree.*

Note. Typically, the initial MST has very few nodes with degree greater than 3 [1]. In practice, it is worth modifying the algorithm to scan the vertices in preorder, maintaining the partial tree T_3 of edges added so far, and to add paths to T_3 as follows. When considering a vertex v, if the degree of v in the partial T_3 is 2, add the path P_v as described in the algorithm. Otherwise, its degree is 1, so, in this case, relax the requirement that the added path must start at v. That is, add the shortest path that visits v and all of v's children to T_3 (see §3.3). This modification will never increase the cost of the resulting tree but may offer substantially lighter trees in practice.

LEMMA 3.4. *The algorithm in Fig.* 2 *outputs a spanning tree of degree* 3.

Proof. An easy proof by induction shows that the union of the paths forms a tree. Each vertex v is on at most two paths and is an interior vertex of at most one path. □

LEMMA 3.5. *Let v be a vertex in an MST T of a set of points in $\Re^2$. Let P_v be a shortest path visiting $\{v\} \cup child_T(v)$ with v as one of its endpoints.*

$$w(P_v) \leq 1.5 \times \sum_{v_i \in child_T(v)} \overline{vv_i}.$$

By the above lemma, each path P_v has weight at most 1.5 times the weight of the edges it replaces. Thus we have the following theorem.

THEOREM 3.6. *Let T be an MST of a set of points in $\Re^2$. Let T_3 be the spanning tree output by the algorithm in Fig.* 2.

$$w(T_3) \leq 1.5 \times w(T).$$

Proof of Lemma 3.5. We consider the various cases that arise depending on the number of children of v. The cases when v has no children or exactly one child are trivial.

Case 1. *v has* 2 *children, v_1, v_2.* There are two possible paths for P_v, namely $P_1 = [v, v_1, v_2]$ and $P_2 = [v, v_2, v_1]$. Clearly,

$$w(P_v) = \min(w(P_1), w(P_2)) \leq \frac{w(P_1) + w(P_2)}{2} = \frac{\overline{vv_1}}{2} + \frac{\overline{vv_2}}{2} + \overline{v_1v_2} \leq 1.5\ (\overline{vv_1} + \overline{vv_2}).$$

Case 2. *v has* 3 *children, v_1, v_2, v_3.* Let v_1 be the child that is nearest to v. Consider the following four paths (see Fig. 3): $P_1 = [v, v_1, v_2, v_3], P_2 = [v, v_1, v_3, v_2], P_3 = [v, v_2, v_1, v_3]$, and $P_4 = [v, v_3, v_1, v_2]$.

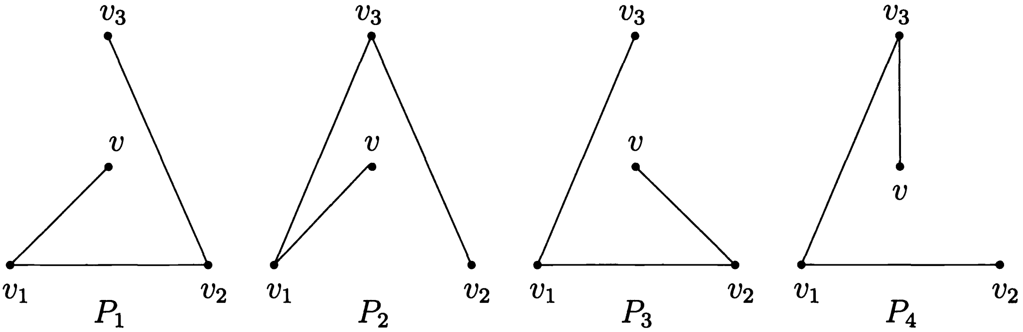


FIG. 3. *T_3, three children.*

The path P_v is at most as heavy as the lightest of $\{P_1, P_2, P_3, P_4\}$. The weight of the lightest of these paths is at most any convex combination of the weights of the paths. Specifically,

$$w(P_v) \leq \min(w(P_1), w(P_2), w(P_3), w(P_4)) \leq \frac{w(P_1)}{3} + \frac{w(P_2)}{3} + \frac{w(P_3)}{6} + \frac{w(P_4)}{6}.$$

We will now prove that

$$\frac{w(P_1)}{3} + \frac{w(P_2)}{3} + \frac{w(P_3)}{6} + \frac{w(P_4)}{6} \leq 1.5\ (\overline{vv_1} + \overline{vv_2} + \overline{vv_3}).$$

This simplifies to

$$\overline{v_1v_2} + \overline{v_2v_3} + \overline{v_3v_1} \leq 1.25\ \overline{vv_1} + 2(\overline{vv_2} + \overline{vv_3}),$$

which follows from Lemma 3.3.

Case 3. *v has 4 children, v_1, v_2, v_3, v_4, ordered clockwise around v.* Let v' be the point of intersection of the diagonals $\overline{v_1v_3}$ and $\overline{v_2v_4}$. Note that the diagonals do intersect because the polygon $v_1v_2v_3v_4$ is convex (follows from Corollary 3.2).

Let v_3 be the point that is furthest from v', among $\{v_1, v_2, v_3, v_4\}$. Consider the following two paths (see Fig. 4): $P_1 = [v, v_4, v_1, v_2, v_3], P_2 = [v, v_2, v_1, v_4, v_3]$.

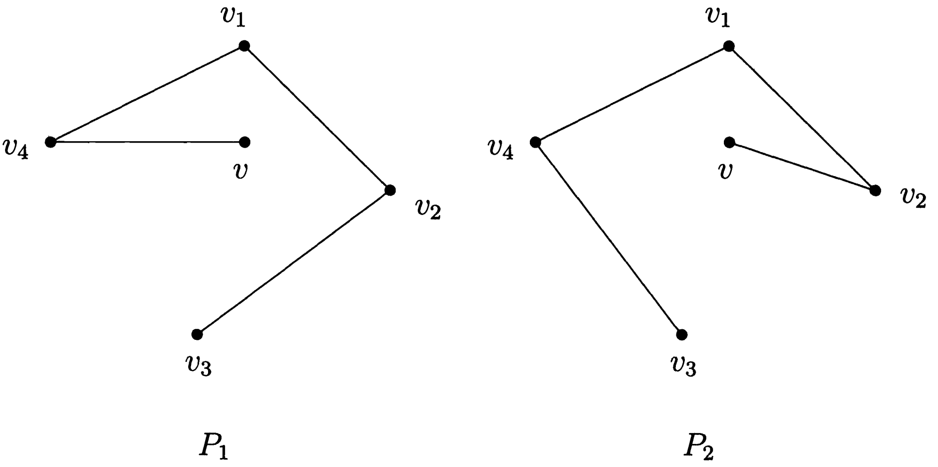


FIG. 4. T_3, *four children.*

Clearly,

$$w(P_v) \leq \min(w(P_1), w(P_2)) \leq \frac{w(P_1)}{2} + \frac{w(P_2)}{2}.$$

We will show that

$$\frac{1}{2}(w(P_1) + w(P_2)) \leq 1.5(\overline{vv_1} + \overline{vv_2} + \overline{vv_3} + \overline{vv_4}).$$

This simplifies to

$$\overline{v_1v_2v_3v_4} + (\overline{v_1v_2} + \overline{v_1v_4}) \leq 3(\overline{vv_1} + \overline{vv_3}) + 2(\overline{vv_2} + \overline{vv_4}). \tag{5}$$

We will first prove that

$$\overline{v_1v_2v_3v_4} + (\overline{v_1v_2} + \overline{v_1v_4}) \leq 3(\overline{v'v_1} + \overline{v'v_3}) + 2(\overline{v'v_2} + \overline{v'v_4}). \tag{6}$$

Once we prove (6), by the triangle inequality, we can conclude that (5) is true, since $\overline{vv_1} + \overline{vv_3} \geq \overline{v_1v_3} = \overline{v'v_1} + \overline{v'v_3}$ and $\overline{vv_2} + \overline{vv_4} \geq \overline{v_2v_4} = \overline{v'v_2} + \overline{v'v_4}$.

We prove (6) by contradiction. Suppose there exists a set of points which does not satisfy (6). Suppose we shrink $v'v_3$ by δ. The left side of the above inequality decreases by at most 2δ, whereas the right side of the inequality decreases by exactly 3δ. Therefore, as we shrink $v'v_3$, the inequality stays violated. Suppose $v'v_3$ shrinks and becomes equal to another edge $v'v_i$ for some $i \in \{1, 2, 4\}$. We now shrink both $v'v_3$ and $v'v_i$ simultaneously at the same rate. Again, it is easy to show that the inequality continues to be violated as $v'v_3$ and $v'v_i$ shrink. Hence we reach a configuration where three of the edges are equal.

Without loss of generality, the length of the three edges is 1 and the length of the fourth edge is some $\epsilon \leq 1$.

There are two subcases to consider. The first is when $v'v_1 = \epsilon$ and the second is when $v'v_2 = \epsilon$. (The case when $v'v_4 = \epsilon$ is the same as the second case.)

Case 3a. $v'v_1 = \epsilon$. We wish to prove that

$$\overline{v_1v_2v_3v_4} + (\overline{v_1v_2} + \overline{v_1v_4}) \leq 7 + 3\epsilon.$$

We want to show that the function $F(\epsilon) = \overline{v_1v_2v_3v_4} + (\overline{v_1v_2} + \overline{v_1v_4}) - 7 - 3\epsilon$ is nonpositive in the domain $0 \leq \epsilon \leq 1$. Simplifying, we get

$$F(\epsilon) = 2\overline{v_1v_2} + \overline{v_2v_3} + \overline{v_3v_4} + 2\overline{v_1v_4} - 7 - 3\epsilon.$$

Each of $\overline{v_iv_j}$ in the definition of F is a convex function of ϵ due to the following reason. Let p be the point closest to v_j on the line connecting v_i and v'. Observe that as v_i moves towards v', $\overline{v_iv_j}$ decreases if v_i is moving towards p and increases otherwise. Since F is a sum of convex functions minus a linear function, it is a convex function of ϵ. Therefore, it is maximized at either $\epsilon = 0$ or $\epsilon = 1$.

When $\epsilon = 1$, all four points are at the same distance from v'. If angle $\angle v_4v'v_1 = \alpha$ then F can be written as a function of a single variable α and it can be verified that F reaches a maximum value of $10\sqrt{0.8} - 10$, which is nonpositive.

When $\epsilon = 0$, $\overline{v_1v_2} = \overline{v_1v_4} = 1$. Simplifying we get $F = \overline{v_2v_3} + \overline{v_3v_4} - 3$, and it reaches a maximum value of $2\sqrt{2} - 3$, which is nonpositive (when $\epsilon = 0$, note that v_1 is the midpoint of the line segment v_2v_4).

Case 3b. $v'v_2 = \epsilon$. We wish to prove that

$$\overline{v_1v_2v_3v_4} + (\overline{v_1v_2} + \overline{v_1v_4}) \leq 8 + 2\epsilon.$$

We want to show that the function $F'(\epsilon) = \overline{v_1v_2v_3v_4} + (\overline{v_1v_2} + \overline{v_1v_4}) - 8 - 2\epsilon$ is nonpositive in the domain $0 \leq \epsilon \leq 1$.

As a function of ϵ, function F' is a sum of convex functions minus a linear function and thus is convex. Therefore, it is maximized at either $\epsilon = 0$ or $\epsilon = 1$.

The case $\epsilon = 1$ leads to the same configuration as in Case 3a.

When $\epsilon = 0$, $\overline{v_1v_2} = \overline{v_2v_3} = 1$. Here $F' = 2\overline{v_1v_4} + \overline{v_3v_4} - 5$. If angle $\angle v_4v'v_1 = \alpha$, then F' can be written as a function of a single variable α and it can be verified that F' reaches a maximum value of $5\sqrt{0.8} - 5$, which is nonpositive.

This concludes the proof of Lemma 3.5. □

The example in Fig. 5 shows that the 1.5 factor is tight for the algorithm in Fig. 2, modified according to the note following its description. The same example also shows that the 1.5 factor is tight for the unmodified algorithm since the unmodified algorithm never outputs a lighter tree than the modified algorithm. Each curved arc shown in Fig. 5 is actually a straight line and has been drawn curved for convenience. The vertex that is the child of the root has three children and is forced to drop one child. In doing so, the degree of its child goes to 4, and it in turn drops one of its children. The algorithm could make choices in such a way that the changes propagate through the tree and the tree T_3 output by the algorithm may be as shown in the figure. The ratio of the cost of the final solution to the cost of the MST can be made arbitrarily close to 1.5. See §5 for a discussion on the worst-case ratio between degree-3 trees and MSTs.

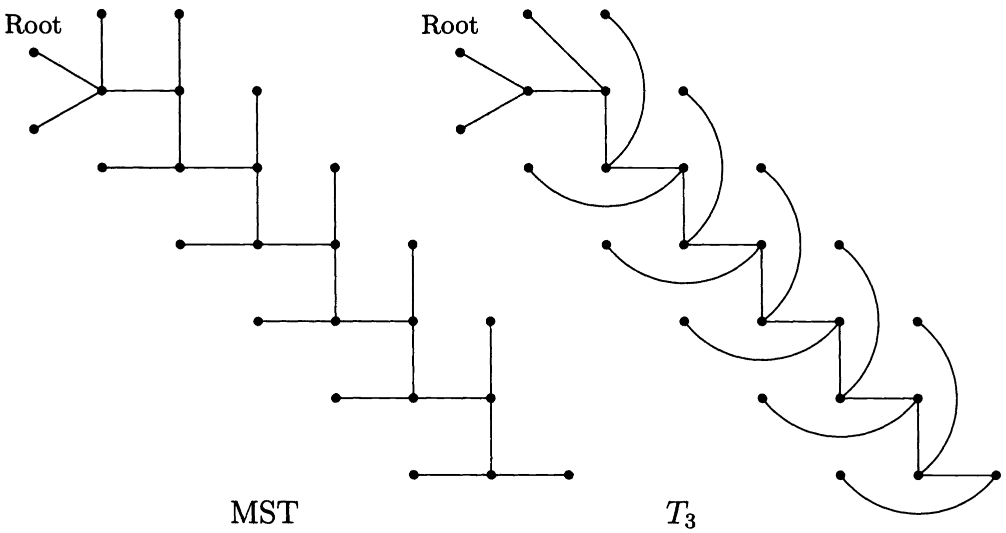


FIG. 5. *Bad example for algorithm in Fig.* 2.

3.3. Spanning trees of degree 4. We now assume that we are given a Euclidean minimum-spanning tree in which every vertex has degree at most 5. We show how to convert this tree to a tree in which every vertex has degree at most 4.

High level description. The basic idea is the same as in the previous algorithm. The difference is that we don't insist that each path P_v start at v. The tree is rooted at an arbitrary leaf. For each vertex v, the minimum-weight path P_v visiting v and all of v's children (not necessarily starting at v) is computed. The final tree T_4 consists of the union of the paths $\{P_v\}$. Again, for the analysis we think of each path P_v replacing the edges between v and its children in T.

```
TREE-4($V, T$) — Find a degree-4 tree of $V$.
Root the MST $T$ at a leaf vertex $r$.
For each vertex $v \in V$ do
    Compute the shortest path $P_v$ visiting $v$ and all its children.
Return $T_4$, the tree formed by the union of the paths $\{P_v\}$.
```

FIG. 6. *Algorithm to find a degree-*4 *tree.*

LEMMA 3.7. *The algorithm in Fig.* 6 *returns a degree-*4 *spanning tree of the given set of points* V.

Proof. A proof by induction shows that T_4 is a tree. Each vertex v occurs in at most two paths and thus has degree at most 4. □

LEMMA 3.8. *Let* v *be a vertex in an MST* T *for a set of points in* $\Re^2$. *Let* P_v *be the shortest path visiting* $\{v\} \cup \mathit{child}_T(v)$.

$$w(P_v) \leq 1.25 \times \sum_{v_i \in \mathit{child}_T(v)} \overline{vv_i}.$$

From the above lemma, each path P_v weighs at most 1.25 times the net weight of the edges it replaces. Thus we have the following theorem.

THEOREM 3.9. *Let T be an MST of a set of points in $\Re^2$. Let T_4 be the spanning tree output by the algorithm in Fig.* 6.

$$w(T_4) \leq 1.25 \times w(T).$$

Proof of Lemma 3.8. The proof is similar to the proof of Lemma 3.5. As before, we consider cases depending on the number of children of v. The cases when v has no children, one child, or two children are trivial.

Case 1. *v has 3 children, v_1, v_2, v_3.* Let v_1 be the point that is closest to v, among its children. Consider the following four paths (see Fig. 7): $P_1 = [v_2, v_1, v, v_3], P_2 = [v_2, v, v_1, v_3], P_3 = [v_1, v, v_2, v_3]$, and $P_4 = [v_1, v, v_3, v_2]$.

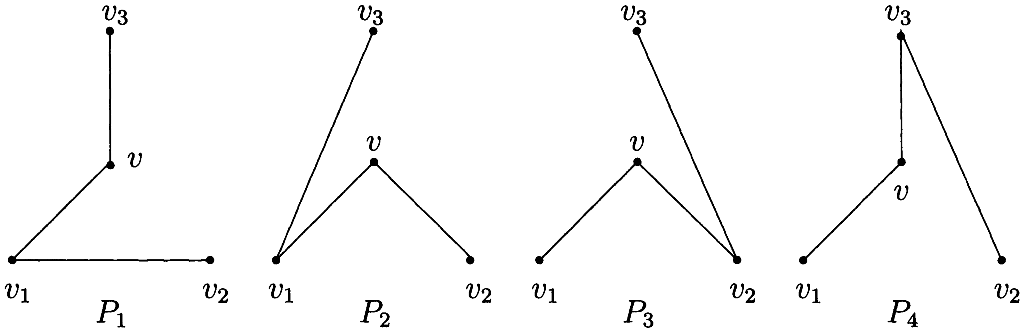


FIG. 7. *T_4, three children.*

Clearly,

$$w(P_v) \leq \frac{w(P_1)}{3} + \frac{w(P_2)}{3} + \frac{w(P_3)}{6} + \frac{w(P_4)}{6}.$$

We will show that

$$\frac{w(P_1)}{3} + \frac{w(P_2)}{3} + \frac{w(P_3)}{6} + \frac{w(P_4)}{6} \leq \frac{2+\sqrt{3}}{3}(\overline{vv_1} + \overline{vv_2} + \overline{vv_3}).$$

This proves the three-child case because $\frac{2+\sqrt{3}}{3}$ approximately equals 1.244 and is less than 1.25. This simplifies to

$$\frac{\overline{v_1v_2} + \overline{v_1v_3} + \overline{v_2v_3}}{3} + \overline{vv_1} + \frac{\overline{vv_2} + \overline{vv_3}}{2} \leq \frac{2+\sqrt{3}}{3}(\overline{vv_1} + \overline{vv_2} + \overline{vv_3}),$$

which further simplifies to

$$\overline{v_1v_2v_3} \leq (\sqrt{3}-1)\overline{vv_1} + \left(\sqrt{3} + \frac{1}{2}\right)(\overline{vv_2} + \overline{vv_3}). \tag{7}$$

Since v_1 is the closest point to v, applying Lemma 3.3, we get

$$\overline{v_1v_2v_3} \leq (3\sqrt{3}-4)\overline{vv_1} + 2(\overline{vv_2} + \overline{vv_3}),$$

and hence

$$\begin{aligned}\overline{v_1v_2v_3} &\le (\sqrt{3}-1)\overline{vv_1} + (2\sqrt{3}-3)\overline{vv_1} + 2(\overline{vv_2} + \overline{vv_3})\\ &\le (\sqrt{3}-1)\overline{vv_1} + (\sqrt{3}+\frac{1}{2})(\overline{vv_2}+\overline{vv_3}).\end{aligned}$$

This proves (7).

Case 2. *v has* 4 *children,* v_1, v_2, v_3, v_4. Assume that v_1 is the point that is closest to v among v's children. Let the order of the points be v_1, v_2, v_3, v_4 when we scan the plane clockwise from v starting from an arbitrary direction.

There are two cases, depending on whether v_4 or v_3 is the point that is furthest from v among its children. We first address the case when v_4 is the furthest point. (The proof for the case when v_2 is the point furthest from v is symmetric to the case when v_4 is the furthest point.)

Consider the following paths: $P_1 = [v_4, v_1, v, v_2, v_3]$ and $P_2 = [v_4, v_3, v, v_1, v_2]$ (see Fig. 8).

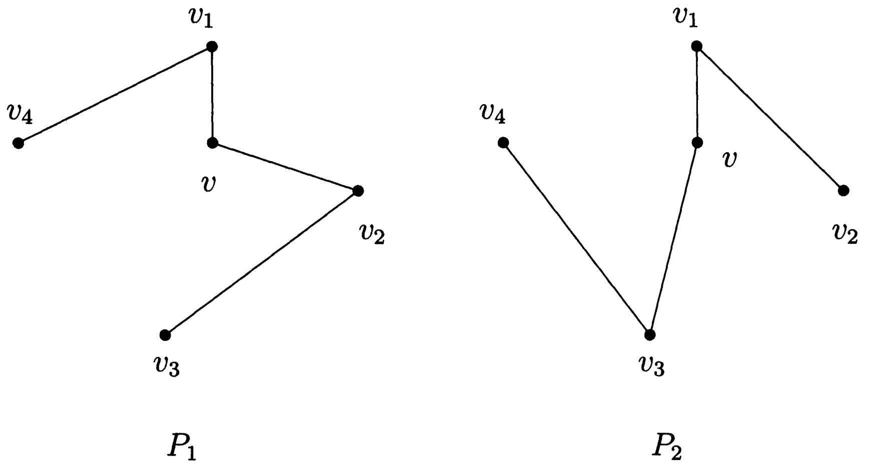


FIG. 8. T_4, *four children.*

The path P_v added by the algorithm is at most as heavy as the lighter of the paths P_1 and P_2. Hence

$$w(P_v) \le \min(P_1, P_2) \le \frac{w(P_1)+w(P_2)}{2}.$$

We will show that

$$\frac{w(P_1)+w(P_2)}{2} \le 1.25(\overline{vv_1} + \overline{vv_2} + \overline{vv_3} + \overline{vv_4}).$$

Simplifying, we need to show that

$$\frac{1}{2}(\overline{v_4v_1} + \overline{v_1v} + \overline{vv_2} + \overline{v_2v_3} + \overline{v_4v_3} + \overline{v_3v} + \overline{vv_1} + \overline{v_1v_2}) \le \frac{5}{4}(\overline{vv_1} + \overline{vv_2} + \overline{vv_3} + \overline{vv_4}).$$

Further simplifying, we get

$$\overline{v_1v_2v_3v_4} \le \frac{1}{2}\overline{vv_1} + \frac{5}{2}\overline{vv_4} + \frac{3}{2}(\overline{vv_2} + \overline{vv_3}).$$

Note that if it happens that v_3 was the farthest point from v among v's children, we get a similar equation with v_3 and v_4 being exchanged in the r.h.s of the equation. By symmetry, the case when v_2 is furthest is similar to v_4 being farthest.

Without loss of generality, $\overline{vv_3} \geq \overline{vv_2}$. The proof now proceeds in a manner similar to the proof of Lemma 3.3. If there is a configuration of points for which this equation is not true (the l.h.s exceeds the r.h.s) then we can move v_4, v_3 closer to v until $\overline{vv_2} = \overline{vv_3} = \overline{vv_4}$. In doing this, we decrease the l.h.s by at most $2(\overline{vv_4} - \overline{vv_2}) + 2(\overline{vv_3} - \overline{vv_2})$. Clearly, the r.h.s decreases by exactly $4(\overline{vv_4} - \overline{vv_2}) + 4(\overline{vv_3} - \overline{vv_2})$. This ensures that the l.h.s is still greater than the r.h.s. Hence without loss of generality, if there is a configuration for which our equation is not true, then there is a configuration with the property that $\overline{vv_4} = \overline{vv_3} = \overline{vv_2}$. We now show that when this property is true there is no counterexample.

By scaling, we may assume that $\overline{vv_4} = \overline{vv_3} = \overline{vv_2} = 1$ and $\overline{vv_1} = \epsilon$, where $\epsilon \leq 1$.

Note that (by Corollary 3.2) v was originally within the convex hull of its four children. Also (by Corollary 3.2), every child is on the convex hull. These properties are both maintained by the above shrinking steps.

We now wish to prove that

$$\overline{v_1v_2v_3v_4} \leq \frac{11}{2} + \frac{1}{2}\epsilon.$$

It is easily shown using elementary calculus that for any ϵ such that v_1 is on the convex hull of the points $\{v_1, \ldots, v_4\}$, rotating v_1 and v_3 around v until $\angle v_1vv_2 = \angle v_1vv_4$ (see Fig. 9) and $\angle v_2vv_3 = \angle v_4vv_3$ does not decrease the perimeter. Also, it maintains that v_1 is on the convex hull. Assume the two pairs of angles are equal, and define $F(\epsilon) = \overline{v_1v_2v_3v_4} - \epsilon/2 - 11/2$. We will show that F is nonpositive over the domain of possible ϵ's.

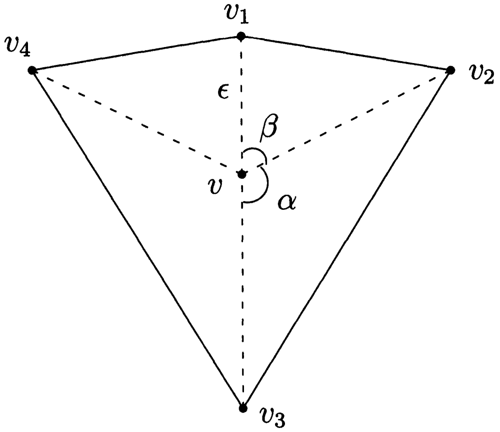


FIG. 9. *Figure to illustrate degree-4 case.*

As a function of ϵ, function F is a sum of convex functions minus a linear function and thus is convex. Therefore, F is maximized either when $\overline{vv_1} = 1$ or when v_1 is the midpoint of edge $\overline{v_2v_4}$ (since v_1 is on the convex hull, v_1 cannot cross the edge; hence this interval contains all possible values for ϵ).

In the first case, all four points lie on a unit circle with center at v. For any four such points, it is easily proven using calculus that $\overline{v_1v_2v_3v_4}$ is maximized when the four points are the vertices of a square at $4\sqrt{2} \approx 5.66$. Thus, $F(1) < 0$.

In the second case, $\overline{v_1v_2v_3v_4} = \overline{v_2v_3v_4}$. As noted previously, this is at most $3\sqrt{3} \approx 5.2$. Thus, $F(\epsilon) < 0$.

We now deal with the case when v_3 is the furthest point. In this case, we take the paths $P_1 = [v_4, v_1, v, v_2, v_3]$ and $P_2 = [v_3, v_4, v, v_1, v_2]$. The path P added by the algorithm is at most as heavy as the lighter of the paths P_1 and P_2. Hence,

$$w(P) \leq \min(P_1, P_2) \leq \frac{w(P_1) + w(P_2)}{2}.$$

Simplifying, we get

$$\overline{v_1v_2v_3v_4} \leq \frac{1}{2}\overline{vv_1} + \frac{5}{2}\overline{vv_3} + \frac{3}{2}(\overline{vv_2} + \overline{vv_4}).$$

The proof of this is identical to the proof of the previous case. □

4. Points in higher dimensions. We show how to compute a degree-3 tree (T_3) when the points are in arbitrary dimension $d \geq 3$. The algorithm for computing the tree is similar to the algorithm for computing degree-3 trees in the plane—the tree T_3 is formed by rooting the MST and taking the union of the paths $\{P_v\}$, where each P_v is the shortest path starting at v and visiting all of the children of v in the rooted MST. It is known that any Euclidean MST has constant degree [17] (for any fixed dimension), so that the algorithm still requires only linear time. The bound on the weight of T_3 is similar, except that v may have more children. We prove that regardless of the number of children that v has, the weight of P_v is at most 5/3 the weight of the edges that it replaces:

LEMMA 4.1. *Let $\{v, v_1, v_2, \ldots, v_k\}$ be a set of arbitrary points in $\Re^d$. There is a path P, starting at v, that visits all the points $v_1, v_2, \ldots, v_k$ such that*

$$w(P) \leq \frac{5}{3}\sum_{i=1}^{k}\overline{vv_i}.$$

Proof. We prove this by induction on the degree of v. Sort the points in increasing distance from v as $v_1, \ldots, v_k$. Let $v = v_0$. The lemma is trivially true when $k = 0, 1, 2$. Let us assume that the lemma is true for all values of k up to some $\ell \geq 2$. Consider $k = \ell + 1$. By the induction hypothesis, the claim is true when v has $k - 3$ children; hence we can find a path P' that starts at v and visits all vertices v_i ($i = 1, \ldots, k-3$) (not necessarily in that order) such that $w(P') \leq (5/3)\sum_{i=1}^{k-3}\overline{vv_i}$. Let v_j be the last vertex on the path P'. We add the cheapest path P'' that starts at v_j and visits v_{k-2}, v_{k-1}, and v_k (again, not necessarily in that order). This path together with P' will form a path that starts at v and visits all vertices adjacent to v. We now show that

$$w(P'') \leq \frac{5}{3}(\overline{vv_{k-2}} + \overline{vv_{k-1}} + \overline{vv_k}). \tag{8}$$

This suffices to prove the lemma. Let $P_1, \ldots, P_6$ be the six possibilities for P''. Clearly,

$$w(P'') \leq \frac{1}{6}\sum_{i=1}^{6} w(P_i).$$

We will prove that

$$\frac{1}{6}\sum_{i=1}^{6} w(P_i) \leq \frac{5}{3}(\overline{vv_{k-2}} + \overline{vv_{k-1}} + \overline{vv_k}).$$

This simplifies to

$$(9) \qquad 2\,\overline{v_{k-2}v_{k-1}v_k} + \sum_{i=k-2}^{k} \overline{v_j v_i} \leq 5(\overline{vv_{k-2}} + \overline{vv_{k-1}} + \overline{vv_k}).$$

Notice that if the above equation is not true, we can "shrink" all the v_i $(i = k-2, k-1, k)$ until $\overline{vv_j} = \overline{vv_{k-2}} = \overline{vv_{k-1}} = \overline{vv_k}$. Assume that $\delta = (\overline{vv_{k-2}} - \overline{vv_j}) + (\overline{vv_{k-1}} - \overline{vv_j}) + (\overline{vv_k} - \overline{vv_j})$. This can be done because the r.h.s decreases by 5δ and the l.h.s decreases by at most 5δ. If the above equation is not true, then it is also not true when the distance from v to all the points is the same. By scaling, we can assume that the distance of the points from v is 1. We call this a canonical configuration. The following proposition is implied by Lillington's work [13] and helps in completing the proof.

PROPOSITION 4.2. *Let $A, B, C,$ and D be points on a unit sphere in d dimensions, $d \geq 3$. The function $F = \overline{AB} + \overline{AC} + \overline{AD} + \overline{BC} + \overline{CD} + \overline{BD}$ reaches a maximum value of $4\sqrt{6}$ when the points $A, B, C,$ and D form a regular tetrahedron.*

We will now show that (9) is satisfied by the canonical configuration. The left side of (9) can be written as the sum of the sides of the tetrahedron formed by the points $\{v_k, v_{k-1}, v_{k-2}, v_j\}$ and the sum of the sides of the triangle formed by the points $\{v_k, v_{k-1}, v_{k-2}\}$. These points lie on a sphere whose center is v. By Lemma 4.2, the first sum is bounded by $4\sqrt{6}$. The second sum is bounded by $3\sqrt{3}$. Hence the left side of (9) is bounded by $4\sqrt{6} + 3\sqrt{3}$, which is about 14.994. The right side of (9) is 15. Hence (9) is satisfied by the canonical configuration and therefore all configurations. This concludes the proof of Lemma 4.1. □

Remark. The algorithm outlined earlier runs in linear time only when d, the number of dimensions, is a constant. The algorithm can be modified to run in linear time for all d as follows. Observe that in the proof of Lemma 4.1, we considered the neighbors of v only three at a time. Therefore, the algorithm could also group vertices into sets of three each, based on the distance from v, and inductively construct the path as in the proof of the lemma. This algorithm would have the same performance guarantee (5/3) as the earlier algorithm for constructing a degree-3 tree and in addition have the added advantage of running in linear time for all dimensions.

5. Conclusions. We have given a simple algorithm for computing a degree-3 (degree-4) tree for points in the plane that is within 1.5 (1.25) of an MST of the points. An extension of the algorithm finds a degree-3 tree of an arbitrary set of points in d dimensions within 5/3 of an MST. If an MST of the points is given as part of the input, our algorithms run in linear time. All our proofs are based on elementary geometric techniques.

Though our algorithms improve greatly the best-known ratios for each of the respective problems, there are still large gaps between the ratios that we obtain and the best bounds that we think are achievable. For example, in the case of points in the plane, consider the ratio of the weight of a minimum weight degree-3 tree to the weight of an MST. The worst example that we can obtain for this ratio is $\frac{\sqrt{2}+3}{4} \approx 1.104$ (with five points, where four of the points are at the corners of a square and the fifth point is in the middle). There is a large gap between this and the ratio of 1.5 obtained by our algorithm. Is 1.104 the worst-case ratio? Are there polynomial time algorithms which obtain factors better than 1.5? Notice that the performance ratio obtained by our algorithm on the example in Fig. 5 is highly sensitive to the vertex chosen as the root. One potential algorithm is to simply try all possible vertices as the root, and to pick the tree of minimum weight. Does such an algorithm have a better performance guarantee?

For the problem of finding degree-4 trees, our algorithm obtains a ratio of 1.25. Unlike degree-3 trees, we are unable to show that this ratio is tight for the algorithm. Can the factor of 1.25 for the algorithm be improved? The worst example for the ratio between a minimum-weight degree-4 tree and an MST that we can obtain is about 1.035 (five points on the vertices of a regular pentagon with a sixth point in their centroid). Are there examples with worse ratios?

Problems of approximating degree-k trees in higher dimensions and in general metric spaces within factors better than 2 are still open.

Acknowledgments. We thank Andras Bezdek for telling us about [13]. We thank Karoly Bezdek and Bob Connelly for useful discussions and the committee members of the 1994 Syposium on the Theory of Computing for simplifying the proof of Lemma 3.3 and for pointing out [14].

REFERENCES

[1] J. L. Bentley, Communicated by David Johnson.

[2] D. Bienstock, M. X. Goemans, D. Simchi-Levi, and D. P. Williamson, *A note on the prize collecting traveling salesman problem*, Math. Programming, 59 (1993), pp. 413–420.

[3] N. Christofides, *Worst-case analysis of a new heuristic for the traveling salesman problem*, Tech. report 388, Graduate School of Industrial Administration, Carnegie Mellon University, Pittsburgh, PA, 1975.

[4] G. Das and P. J. Heffernan, *Constructing degree-3 spanners with other sparseness properties*, in Proc. 4th Annual International Symposium on Algorithms and Computation, Lecture Notes in Comput. Sci. 762, Springer-Verlag, Berlin, New York, 1993, pp. 11–20.

[5] D.-Z. Du and F. K. Hwang, *A proof of the Gilbert–Pollak conjecture on the Steiner ratio*, Algorithmica, 7 (1992), pp. 121–136.

[6] D.-Z. Du, Y. Zhang, and Q. Feng, *On better heuristic for Euclidean Steiner minimum trees*, in Proc. 32nd Annual Symposium on Foundations of Computer Science, IEEE Press, Piscataway, NJ, 1991, pp. 431–439.

[7] T. Fischer, *Optimizing the degree of minimum weight spanning trees*, Tech. report 93-1338, Department of Computer Science, Cornell University, April 1993.

[8] G. N. Frederickson and J. JáJá, *On the relationship between the biconnectivity augmentation and traveling salesman problems*, Theoret. Comput. Sci., 19 (1982), pp. 189–201.

[9] M. Fürer and B. Raghavachari, *Approximating the minimum-degree Steiner tree to within one of optimal*, J. Algorithms, 17 (1994), pp. 409–423.

[10] M. R. Garey and D. S. Johnson, *Computers and intractability: A guide to the theory of NP-completeness*, W. H. Freeman, San Francisco, 1979.

[11] M. X. Goemans and D. P. Williamson, *A general approximation technique for constrained forest problems*, in Proc. 3rd Annual ACM–SIAM Symposium on Discrete Algorithms, Society for Industrial and Applied Mathematics, Philadelphia, 1992, pp. 307–316; SIAM J. Comput., 24 (1995), pp. 296–317.

[12] A. Itai, C. H. Papadimitriou, and J. L. Szwarcfiter, *Hamilton paths in grid graphs*, SIAM J. Comput., 11 (1982), pp. 676–686.

[13] J. N. Lillington, *Some extremal properties of convex sets*, Math. Proc. Cambridge Philos. Soc., 77 (1975), pp. 515–524.

[14] C. Monma and S. Suri, *Transitions in geometric minimum spanning trees*, Discrete Comput. Geom., 8 (1992), pp. 265–293.

[15] C. H. Papadimitriou and U. V. Vazirani, *On two geometric problems related to the traveling salesman problem*, J. Algorithms, 5 (1984), pp. 231–246.

[16] R. Ravi, M. V. Marathe, S. S. Ravi, D. J. Rosenkrantz, and H. B. Hunt III, *Many birds with one stone: Multi-objective approximation algorithms*, in Proc. 25th Annual ACM Symposium on the Theory of Computing, Association for Computing Machinery, New York, 1993, pp. 438–447.

[17] G. Robins and J. S. Salowe, *On the maximum degree of minimum spanning trees*, in Proc. 10th Annual ACM Symposium on Computational Geometry, Association for Computing Machinery, New York, 1994, pp. 250–258; Discrete Comput. Geom., 14 (1995), pp. 151–166.

[18] J. S. Salowe, *Euclidean spanner graphs with degree four*, in Proc. 8th Annual ACM Symposium on Computational Geometry, Association for Computing Machinery, New York, 1992, pp. 186–191; Discrete Appl. Math., 54 (1994), pp. 55–66.

SIAM J. COMPUT.
Vol. 25, No. 2, pp. 369–389, April 1996

OPTIMAL CLOCK SYNCHRONIZATION UNDER DIFFERENT DELAY ASSUMPTIONS*

HAGIT ATTIYA[†], AMIR HERZBERG[‡], AND SERGIO RAJSBAUM[§]

Abstract. The problem of achieving optimal clock synchronization in a communication network with arbitrary topology and perfect clocks (that do not drift) is studied. Clock synchronization algorithms are presented for a large family of delay assumptions. Our algorithms are modular and consist of three major components. The first component holds for any type of delay assumptions; the second component holds for a large, natural family of local delay assumptions; the third component must be tailored for each specific delay assumption.

Optimal clock synchronization algorithms are derived for several types of delay assumptions by appropriately tuning the third component. The delay assumptions include lower and upper delay bounds, no bounds at all, and bounds on the difference of the delay in opposite directions. In addition, our model handles systems where some processors are connected by broadcast networks in which every message arrives at all the processors at approximately the same time. A composition theorem allows combinations of different assumptions for different links or even for the same link; such mixtures are common in practice.

Our results achieve the best possible precision in each execution. This notion of optimality is stronger than the more common notion of worst-case optimality. The new notion of optimality applies to systems where the worst-case behavior of any clock synchronization algorithm is inherently unbounded.

Key words. distributed systems, real-time systems, clock synchronization, message passing systems, networks, optimization, message delay assumptions, precision

AMS subject classifications. 68Q10, 68Q22, 68Q25, 68R10

1. Introduction.

In most large-scale distributed systems, processors communicate by message transmission and do not have access to a central clock. Nonetheless it is useful, and sometimes even necessary, for the processors to obtain some common notion of time. The technique used to attain this notion of time is known as *clock synchronization.* Synchronized clocks are useful for various applications such as control of real-time processes, transaction processing in database systems, and communication protocols. Recently, several software protocols that support clock synchronization in communication networks have been proposed [1, 7, 13, 15, 16]; system designers have been advocating the use of synchronized clocks [10].

* Received by the editors April 20, 1994; accepted for publication (in revised form) September 7, 1994.

[†] Department of Computer Science, Technion, Haifa 32000, Israel (hagit@cs.technion.ac.il). The research of this author was supported by grant 92-0233 from the United States–Israel Binational Science Foundation (BSF), Jerusalem, Israel, the Technion V.P.R.—Argentinian Research Fund, and the fund for the promotion of research in the Technion.

[‡] IBM T. J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598 (amir@watson.ibm.com). The research of this author was partially supported by Direccion General de Asuntos del Personal Académico (DGAPA) Projects, National Autonomous University of Mexico (UNAM).

[§] Laboratory for Computer Science, Massachusetts Institute of Technology, 545 Technology Square, Cambridge, MA 02139 (rajsbaum@theory.lcs.mit.edu); on leave from Instituto de Matemáticas, UNAM, Mexico. The research of this author was partially supported by Direccion General de Asuntos del Personal Académico (DGAPA) Projects, UNAM.

The quality of synchronization is measured by its *precision*, i.e., how close together it brings the clocks at different processors.[1] The precision influences the correctness and the efficiency of applications using the synchronized clocks.

The best precision that can be achieved is determined by the timing uncertainty that is inherent in the system. There are two main sources of timing uncertainty in a distributed system. First, local clocks at different processors are independent: they do not start together and may run at different speeds. Second, messages sent between processors incur uncertain delays.

A relatively simple case is when local clocks are accurate, i.e., run at the same speed, and there are upper and lower bounds for the delay on each link. Clock synchronization algorithms under this assumption, whose precision is optimal in the worst case, are described in [4, 11]. Subsequent work concentrated on clocks that may drift and on fault tolerance (e.g., [2, 7, 20, 21]; see the survey in [19]). To achieve high precision, these algorithms require the existence of tight lower and upper bounds on message delay.

However, in real systems it is often the uncertainty of message delay, rather than clock drift, that causes most of the difficulty in synchronizing clocks [13, 7]. Almost every processor in a distributed system has access to a high-quality, very accurate hardware clock; it is not unrealistic to assume that local clocks are accurate and have no drift.[2] On the other hand, often there do not exist tight upper and lower bounds on message delay, while there is other relevant information about the delays. For example, in some systems, a bound on the difference between delays in opposite directions is known. This motivated us to revisit the case in which local clocks run at the same speed and have no drift, thus focusing on the impact of message delay uncertainty on clock synchronization.

Our main contribution is a methodology for designing optimal clock synchronization algorithms under a variety of assumptions on message delay uncertainty. The strongest results are obtained for a natural family of local delay assumptions; this family includes all the assumptions studied in previous theoretical work. A delay assumption is *local* if it is specified for a pair of processors, e.g., processors connected by a link. We show that in this case, a clock synchronization algorithm can be obtained by considering each pair separately. This simplifies the analysis substantially and allows different pairs of processors to satisfy different assumptions. Furthermore, we prove a composition theorem that allows us to combine several local delay assumptions. For example, it is possible that for some pair of processors there will be upper and lower bounds on the message delay in each direction as well as a bound on the difference between message delays in opposite directions.

The basic difficulty of clock synchronization stems from the existence of two different system executions in which all processors have the same views. The tightness of the achievable synchronization depends on how "far away" in real time such executions can be. We capture this notion quantitatively as the *maximal shift* between processors in a given execution. Using this notion, we partition the design of a clock synchronization algorithm into three stages.

[1] This does not ensure that clocks are close to real time. It is easy to adapt our results to reach this goal if a perfect real-time clock is available. Synchronization to real time is often useful and is achieved by practical protocols that usually also deal with multiple, imperfect real-time clocks, e.g., the Internet NTP [13].

[2] To deal with the small drift that does exist, the clock synchronization mechanism is invoked periodically; see, e.g., Kopetz and Ochsenreiter [7].

1. Computing corrections to the local clocks from the maximal shifts. This stage is valid for any kind of message delay assumptions.
2. Computation of maximal shifts from *maximal local shifts*, which depend only on the views of pairs of processors. This stage is valid for message delay assumptions that are local.
3. Computation of the maximal local shifts from the local views. This depends on the specific message delay assumptions.

The computations in stages 1 and 2 are performed by a centralized, off-line algorithm.

Our methodology yields optimal clock synchronization algorithms for a variety of delay assumptions by adapting the third stage. In particular, we show how to compute maximal local shifts for the following message delay assumptions:

1. upper and lower bounds on delays are known (including the degenerated cases of zero lower bound and/or infinite upper bound);
2. there is a bound on the difference on the delay in opposite directions; and
3. there is a bound on the difference in the times when different processors receive a multicast message.

Most previous formal work on deterministic clock synchronization addressed only a restricted version of the first assumption where the delay upper bounds are finite. However, an observation of [1] shows that in many actual links, there is some minimal delay (e.g., due to the actual transmission rate and processing time), while there is no known upper bound. The second assumption follows experimental results (cf. [13]), showing that message delays in opposite directions of a bidirectional link are usually very close. The last assumption is useful for broadcast networks that are used in many local area networks; this is the assumption used in [5, 18].

Our composition theorem implies that our algorithms apply to systems where the same pair of processors satisfies several different delay assumptions. Such mixtures are quite common in practical, heterogeneous systems. For example, there are systems in which several local area (broadcast) networks are connected by bridges or (long-distance) links.

Our work extends the results of Halpern, Megiddo, and Munshi [4], who use linear programming techniques that do not illuminate the inherent difficulties of synchronizing clocks. We believe that our work gives a more precise understanding of the problem, explicitly showing the requirements of each step and thereby facilitating adaptation to other delay assumptions. Their results are a special case of the general methodology developed here, in which exactly one message is sent on each link, and upper and lower bounds on delays are known. In fact, the algorithm we obtain for this specific setting is essentially the one in [4].

Previous definitions of optimal clock synchronization were based on the worst (largest) difference between clocks of two processors in any execution. For some of the assumptions that we study in this paper, e.g., when no upper bounds on the delays are known, this worst case is inherently unbounded. Moreover, as already stated in [4], we would like to award algorithms that exploit favorable conditions and achieve the best possible precision in each specific instance. We give a precise definition of optimality for each specific execution and show that it is achieved by our algorithms (and hence also by the algorithm of [4]).

When trying to crystallize these ideas, it turned out that the decision of which messages to send should be separated from the method for adjusting the clocks based on the local message histories. Our framework shows how to adjust the clocks optimally, given any set of local message histories. The decision of which messages to send, to whom, when, etc., can therefore take other considerations into account, e.g., message traffic optimization, and is left outside of the scope of this paper.

The rest of this paper is organized as follows. In §2 the model and the clock synchronization problem are defined. Section 3 presents the general clock synchronization algorithm and proves that it achieves optimal precision; the algorithm is independent of the message delay assumptions. In §4 we show how to compute the inputs needed for the general clock synchronization algorithm, when some local information about the views of the processors is given; the computation is valid for local systems. In §5 we show how to compute the required information on the views for several specific delay assumptions. Conclusions and open questions appear in §6.

2. Definitions.

2.1. Defining optimal precision. We would like a clock synchronization algorithm to obtain the best possible precision, that is, to bring the logical clocks as close to each other as possible. However, it is not obvious how to compare the precision achieved by different algorithms and how to define optimality.

An elegant solution is to evaluate a clock synchronization algorithm by the worst (largest) precision achieved in any of its executions. This worst-case interpretation follows the tradition of worst-case complexity analysis of algorithms.

This definition has two drawbacks. First, like any definition that concentrates on the worst case, it does not award algorithms that behave well in other cases. An algorithm that is optimal under this definition can be very inefficient in executions where the delays are favorable. Second, worst-case analysis is meaningful only if the worst-case precision is bounded. However, in many important cases, the worst-case precision can easily be shown to be unbounded, e.g., when there are no upper bounds on message delay.

We believe a more refined notion of optimality is warranted. Intuitively, an optimal algorithm is one whose precision, in every execution, is not larger than the precision of any other algorithm in an execution where the message delivery system "acts the same."

Formalizing this idea is not so simple. The major difficulty is finding a satisfying definition for executions where the message delivery system acts the same. The problem is that the properties of the execution are determined by the interaction between the message delivery system and the algorithm. The algorithm controls the execution by deciding when to send messages, while the message delivery system controls the execution by determining their delay.[3] It is difficult to isolate the effect on the execution determined by the message delivery system. Such isolation is necessary in order to compare executions of a given algorithm to executions of other algorithms where the message delivery system is equally adversarial. A definition is too strong if it compares an execution of one algorithm with an execution of another algorithm in which message delays are unfairly favorable. Conversely, a definition is too weak if executions with the same message delivery policy are not compared. We sidestep this problem by noticing that the construction of a clock synchronization algorithm has two aspects: first, the design of the interactive part where the processors send messages; and second, calculating corrections using the views of the processors that were obtained during the interactive part. In this paper, we do not address the first

[3] This is not merely a formal issue: from a practical point of view, if an algorithm sends too many messages in a short period of time, the network becomes congested and delays are long and highly variant.

aspect. We assume that we have a set of views, one for each processor, and we ask how to compare optimal corrections for this set of views.

2.2. Model of computation. Here we formalize the behavior of the interactive part of a clock synchronization algorithm, which is a distributed algorithm running on a network. The distributed algorithm decides when to send messages, while the network decides when to deliver the messages. The interplay between the distributed algorithm and the network generates a set of executions. The result of this execution is the input to the clock synchronization function.

We consider a set $P = \{p_1, \ldots, p_n\}$ of *processors.* With each processor $p \in P$ we associate a (local) clock. The clock cannot be modified by the processor. Processors do not have access to real time; each processor obtains its only information about time from its clock and from messages sent by other processors. The clock is represented by a local time component, which is a real number. In the sequel, the term *clock time* refers to the local time component of the processor, while the term *real time* refers to the absolute time as measured by an outside observer. In this work we assume that clocks do not drift, i.e., that they run at the same rate as real time, but they are not necessarily synchronized with each other.

We list the *events* that can occur at processor p, together with an informal explanation.

Message-receive events—receive(p, m, q) for all messages m and processors q: processor p receives message m from processor q.

Message-send events—send$(p,\ m,\ q)$ for all messages m and processors q: processor p sends message m to processor q.

Timer-set events—timer-set$(p,\ T)$ for all clock times T: processor p sets a timer to go off when its clock reads T.

Timer events—timer$(p,\ T)$ for all clock times T: a timer that was set for time T on p's clock goes off.

Start events—start$(p,\ 0)$: p starts executing the algorithm, with the initial value of its clock being 0.

The message-receive, timer, and start events are *interrupt events.*

Each processor is modeled as an automation with a (possibly infinite) set of states, including an initial state, and a transition function. Each interrupt event causes an application of the transition function, which runs from states, clock times, and interrupt events to states, sets of message-send events, and sets of timer-set events (for subsequent clock times). That is, the transition function takes as input the current state, clock time, and interrupt event (which is the receipt of a message from another processor or a timer going off) and produces a new state, a set of messages to be sent, and a set of timers to be set for the future.

A *step* of p is a tuple (s, T, i, s', M, TS), where s and s' are states; T is a clock time; i is an interrupt event; M is a set of message-send events; TS is a set of timer-set events; and s', M, and TS are the results of p's transition function acting on s, T, and i. A *history* π of a processor p is a mapping associating to each number from $\Re$ (real time) a finite sequence (possibly empty) of steps such that the following hold.

1. For each real time t, there is only a finite number of times $t' < t$ such that the corresponding sequence of steps is nonempty (thus the concatenation of all the sequences in real-time order is a sequence).
2. The interrupt event in the first step of the history is a start event, and the old state in the first step is p's initial state; furthermore, there are no other

start events; let S_π be the real time of the start event.

3. The old state of each subsequent step is the new state of the previous step.
4. For each real time t, the clock-time component T of each step in the corresponding sequence is equal to $t - S_\pi$ (thus, the clock time of the start event of p is 0).
5. For each real time t, in the corresponding sequence there is at most one timer event and it is ordered after all other events.
6. A timer is received by p at clock time T if and only if p has previously set a timer for T.

An *execution* is a set of histories, one for each processor p in P, such that there is a one-to-one correspondence between the messages received by q from p and the messages sent by p to q for any processors p and q. (To simplify our discussion, we assume that messages are unique, so this correspondence is uniquely defined.) Note that this definition allows messages to be lost and delivered in non-FIFO (first in–first out) order; however, it assumes messages are not corrupted or duplicated. We use the message correspondence to define the *delay* of a message m received in execution α, denoted $\mathrm{d}_\alpha(m)$, to be the real time of receipt minus the real time of sending. When α is clear from the context, we simply write $\mathrm{d}(m)$.

Let $S_{\alpha,p} = S_\pi$ where π is p's history in α; that is, $S_{\alpha,p}$ is the real time of the start event of the processor p in α.

Note that the message delivery system is not explicitly modeled. The requirements from an execution state are that messages are delivered without duplication and that the system does not generate messages; the system can reorder or lose messages.

A system $(P, \mathcal{A})$ is a set of processors P and a set of executions $\mathcal{A}$, called *admissible executions.* For example, $\mathcal{A}$ may allow communication only between specific pairs of processors connected by a link.

The cornerstone of our definitions and proofs is the notion of equivalent executions. Informally, two executions are equivalent if they are indistinguishable to the processors; only an outside observer who has access to the real time can tell them apart.

To formalize this notion, define the *view* of processor p in history π to be the concatenation of the sequences of steps in π in real-time order. (Note that the view includes the clock times.) The real times of occurrence are not represented in the view. Let α be an execution, and let π be p's history in α. The view of p in α is the view of p in π and is denoted $\alpha|p$. Two executions α and α' are *equivalent*, denoted $\alpha \equiv \alpha'$, if for every processor $p \in P, \alpha|p = \alpha'|p$.

2.3. The clock synchronization problem. The goal of a clock synchronization algorithm is to bring the clocks of the processors as close to each other as possible, while keeping the clocks' values with the progress of real time. Intuitively, each processor maintains a *logical clock*, which "corrects" the value of the local clock. Since the logical clock is required not to drift from the progress of real time, it is straightforward to see that the logical clock must be the local clock plus some correction factor. Thus, the goal of a clock synchronization algorithm is to compute a correction for each processor such that, for any two processors, the values of the local clocks (at the same real time) plus the respective corrections are close.

Specifically, a *clock synchronization algorithm* is a function from a set of n views to a vector of n real numbers, called *corrections.* Given a clock synchronization algorithm f and an execution α, we abuse notation and denote by $f(\alpha)$ the vector obtained by applying f to the n views in α; we denote by $f(\alpha, p)$ the component of

$f(\alpha)$ that corresponds to p. Since a clock synchronization algorithm depends only on the views, we have the following claim.

CLAIM 2.1. *If $\alpha \equiv \alpha'$, then $f(\alpha) = f(\alpha')$.*

Recall that at any real time t, the clock value of p is $t - S_{\alpha,p}$. Given a clock synchronization function f, the corrected local time of p in α is $t - S_{\alpha,p} + f(\alpha, p)$. Therefore, $|(S_{\alpha,p} - f(\alpha,p)) - (S_{\alpha,q} - f(\alpha,q))|$ is the difference between the corrected local times of p and q in α.

To capture the precision achieved by some vector of corrections $\vec{x} = \langle x_1, \ldots, x_n \rangle$ denote $\rho(\alpha, \vec{x}) = \max_{p,q \in P} |(S_{\alpha,p} - x_p) - (S_{\alpha,q} - x_q)|$. That is, $\rho(\alpha, \vec{x})$ is the largest discrepancy between two clocks of different processors after they are corrected.

Because the computation of the corrections does not distinguish between equivalent executions, we measure the precision for a specific execution α by considering the worst discrepancy achieved for all the executions equivalent to α. Let $\mathcal{A}$ be the set of admissible executions. Formally, for any execution $\alpha \in \mathcal{A}$, the inherent precision achieved by a vector of corrections $\vec{x}$ is

$$\bar{\rho}(\alpha, \vec{x}) = \sup\{\rho(\alpha', \vec{x}) : \alpha' \equiv \alpha \text{ and } \alpha' \in \mathcal{A}\}.$$

DEFINITION 2.1. *A clock synchronization algorithm f* computes optimal corrections *if for every admissible execution α and every vector of corrections $\vec{x}$, $\bar{\rho}(\alpha, f(\alpha)) \leq \bar{\rho}(\alpha, \vec{x})$.*

We call $\bar{\rho}(\alpha, f(\alpha))$ the *precision* of f on α and use the shorthand $\bar{\rho}(\alpha, f)$.

3. A general clock synchronization algorithm. As mentioned before, the basic difficulty of computing corrections is the fact that there may be two admissible executions α and α' in which all processors have the same views. Clearly, the tightness of the achievable synchronization depends on how "far away" in real time α' can be from α. We formally quantify this idea by defining the maximal shift between processors in a given execution. We show that if estimates of the maximal shifts are available, then there exists a function that computes optimal corrections. This is done by showing a lower bound for the precision that depends only on the maximal shifts. Then we show that this bound is tight by presenting a method for computing corrections that achieves this value as its precision. In subsequent sections we show how to estimate the maximal shifts for specific systems.

3.1. Maximal shifts. Consider two equivalent executions α and α'. It follows that for any $p \in P$, the sequence of steps in α' is equal to the sequence of steps in α, except that p executes its steps at different real times. Since the clocks have no drift, it follows that the difference between the real times of occurrence of a step in α and the corresponding step in α' is *fixed*, independently of the step. This implies that α' can be obtained by "shifting" the steps of the processors in α.

In the rest of this section, we formalize this notion of *shifting* and study its properties. This technique was originally introduced by Lundelius and Lynch [11] to prove lower bounds on the precision achieved by clock synchronization algorithms in complete graphs.

Formally, given a history π of processor p and a real number s, a new history $\pi' = \text{shift}(\pi, s)$ is defined by $\pi'(t) = \pi(t + s)$ for all t. That is, all tuples are shifted earlier in π' by s if s is positive and later by $-s$ if s is negative. Clearly, the views do not change with shifting. Furthermore, we have the following result.

LEMMA 3.1 (Lundelius and Lynch [11]). *Let π be a history of processor p, and let s be a real number. Then $\pi' = \text{shift}(\pi, s)$ is a history of p and $S_{\pi'} = S_\pi - s$.*

Let α and α' be two equivalent executions such that each processor $p \in P$ is shifted in α' with respect to (w.r.t.) α by s_p; the *vector of shifts* of α' w.r.t. α is the vector $\vec{s} = \langle s_1, \ldots, s_n \rangle$. That is, execution α' was obtained by replacing p's history in α, π, with $\text{shift}(\pi, s_p)$, for each $p \in P$, and by retaining the same correspondence between message-send and message-receive events. (Technically, the correspondence is redefined so that a pairing in α that involves the event for p at time t, in α' involves the event for p at time $t - s_p$.) We denote α' by $\text{shift}(\alpha, \vec{s})$. Clearly, we have the following claim.

CLAIM 3.2. *Let* $\alpha' = \text{shift}(\alpha, \langle s_1, \ldots, s_n \rangle)$. *For every message* m *received by processor* p *from* q *in* α, $\mathrm{d}_{\alpha'}(m) = \mathrm{d}_\alpha(m) + (s_q - s_p)$.

The following claim follows from the definitions.

CLAIM 3.3. *For any pair of executions* α *and* α', $\alpha \equiv \alpha'$ *if and only if there exists a vector of shifts* $\vec{s}$ *such that* $\alpha' = \text{shift}(\alpha, \vec{s})$.

We now formalize the notion of how "far away" a processor can be shifted w.r.t. another processor. Fix a system $(P, \mathcal{A})$, and let $\alpha \in \mathcal{A}$. We say that s is an *admissible shift of* q *w.r.t.* p *in* α if there exists a vector of shifts $\vec{s} = \langle s_1, \ldots, s_n \rangle$ with $s_q - s_p = s$ such that $\alpha' = \text{shift}(\alpha, \vec{s})$ is in $\mathcal{A}$. Define

$$\mathrm{ms}_\alpha(p, q) = \sup\{s : s \text{ is an admissible shift of } q \text{ w.r.t. } p \text{ in } \alpha\}.$$

This is the *maximal shift* of q w.r.t. p in α; that is, how far away can q be shifted from p while retaining the admissibility of the execution. Since 0 is obviously an admissible shift of q w.r.t. p in α, it follows that $\mathrm{ms}_\alpha(p, q) \geq 0$. Note that in certain cases, e.g., when no message was sent, the maximal shift can be infinite.

CLAIM 3.4. *Let* $\alpha \in \mathcal{A}$ *and let* $\alpha' \equiv \alpha$. *If* $\alpha' \in \mathcal{A}$, *then* $S_{\alpha',p} - S_{\alpha',q} \leq S_{\alpha,p} - S_{\alpha,q} + \mathrm{ms}_\alpha(p, q)$ *for any two processors* p *and* q.

Proof. By Claim 3.3, $\alpha' = \text{shift}(\alpha, \vec{s})$ for some vector of shifts $\vec{s} = \langle s_1, \ldots, s_n \rangle$. Fix a pair of processors p and q. Since $\alpha' \in \mathcal{A}$, it follows that $s_q - s_p \leq \mathrm{ms}_\alpha(p, q)$. The claim follows since $S_{\alpha',p} = S_{\alpha,p} - s_p$ and $S_{\alpha',q} = S_{\alpha,q} - s_q$. □

3.2. The lower bound. Fix a system $(P, \mathcal{A})$, an admissible execution α, and a clock synchronization algorithm f. The following lemma relates the maximal shift and the attainable precision.

LEMMA 3.5. *For any pair of processors* p *and* q, $\bar{\rho}(\alpha, f) \geq S_{\alpha,p} - f(\alpha, p) - S_{\alpha,q} + f(\alpha, q) + \mathrm{ms}_\alpha(p, q)$.

Proof. Let s be an arbitrary admissible shift of q w.r.t. p in α. Let $\alpha' \in \mathcal{A}$ be an execution such that $\alpha' \equiv \alpha$ and q is shifted w.r.t. p by s. Then

$$\begin{aligned}\bar{\rho}(\alpha, f) \geq \rho(\alpha', f(\alpha')) &\geq S_{\alpha',p} - f(\alpha', p) - S_{\alpha',q} + f(\alpha', q) \\ &= S_{\alpha,p} - f(\alpha', p) - S_{\alpha,q} + f(\alpha', q) + s \\ &= S_{\alpha,p} - f(\alpha, p) - S_{\alpha,q} + f(\alpha, q) + s \quad \text{(by Claim 2.1)}.\end{aligned}$$

Since s was chosen arbitrarily,

$$\bar{\rho}(\alpha, f) \geq S_{\alpha,p} - f(\alpha, p) - S_{\alpha,q} + f(\alpha, q) + \mathrm{ms}_\alpha(p, q). \qquad \square$$

The previous lemma implies that if $\mathrm{ms}_\alpha(p, q)$ is infinite for a pair of processors p, q then no clock synchronization algorithm can achieve a finite precision $\bar{\rho}(\alpha, f)$. The expression defined next will turn out to be a lower bound on the precision that can be achieved in α. Let θ be a cyclic sequence of processors, that is, $\theta = p_0, p_1, \ldots, p_{k-1}, p_k$, where $p_k = p_0$; processors p_i and p_{i+1} are not necessarily adjacent in the graph.

Denote $|\theta| = k$ and $\text{ms}_\alpha(\theta) = \sum_{i=0}^{k-1} \text{ms}_\alpha(p_i, p_{i+1})$. Let $A_\alpha(\theta) = \text{ms}_\alpha(\theta)/|\theta|$, and define

$$A_\alpha^{\max} = \max\{A_\alpha(\theta) : \theta \text{ is a cyclic sequence of processors}\}.$$

THEOREM 3.6. *For any clock synchronization algorithm $f, \bar{\rho}(\alpha, f) \geq A_\alpha^{\max}$.*

Proof. Let $\theta = p_0, \ldots, p_k$ be an arbitrary cyclic sequence of processors (where $p_k = p_0$). By Lemma 3.5,

$$\bar{\rho}(\alpha, f) \geq S_{\alpha,p_i} - f(\alpha, p_i) - S_{\alpha,p_{i+1}} + f(\alpha, p_{i+1}) + \text{ms}_\alpha(p_i, p_{i+1})$$

for every $i, 0 \leq i \leq k-1$. Summing over all the consecutive processors in θ, we have

$$k \cdot \bar{\rho}(\alpha, f) \geq \sum_{i=0}^{k-1} [S_{\alpha,p_i} - f(\alpha, p_i) - S_{\alpha,p_{i+1}} + f(\alpha, p_{i+1}) + \text{ms}_\alpha(p_i, p_{i+1})].$$

Clearly,

$$\sum_{i=0}^{k-1} [S_{\alpha,p_i} - f(\alpha, p_i) - S_{\alpha,p_{i+1}} + f(\alpha, p_{i+1})] = 0,$$

and hence,

$$\bar{\rho}(\alpha, f) \geq \frac{1}{k} \sum_{i=0}^{k-1} \text{ms}_\alpha(p_i, p_{i+1}) = A_\alpha(\theta),$$

as needed. □

3.3. The upper bound. Now we show the converse direction, i.e., that there exists a clock synchronization algorithm f with $\bar{\rho}(\alpha, f) = A_\alpha^{\max}$, for every α, provided certain estimates can be computed from the views. By Theorem 3.6, no other clock synchronization algorithm can achieve better precision. Hence our clock synchronization algorithm computes optimal corrections in the sense of Definition 2.1.

Clearly, if the values of $\text{ms}_\alpha(p, q)$ are known, then it is possible to calculate $A_\alpha^{\max}$. As we shall see, computing $A_\alpha^{\max}$ is the crux of computing optimal corrections. However, since the views do not include the actual message delays, it is not clear what the set of equivalent executions is; hence, in general, it is impossible to compute the values of $\text{ms}_\alpha(p, q)$ from the views. Below we show that it suffices to have only estimates on $\text{ms}_\alpha(p, q)$. In the next sections, we show how to obtain these estimates for specific systems.

Define the *estimated maximal global shift* to be $\widetilde{\text{ms}}_\alpha(p, q) = \text{ms}_\alpha(p, q) + S_{\alpha,p} - S_{\alpha,q}$. The next lemma is the key to replacing ms_α with the estimates $\widetilde{\text{ms}}_\alpha$ in the calculation of $A_\alpha^{\max}$. The lemma shows that the maximum average cycle weight with respect to the actual maximal shifts is equal to the maximum average cycle weight with respect to the estimates. Specifically, for any cyclic sequence of processors $\theta = p_0, \ldots, p_k$ (where $p_k = p_0$), let $\widetilde{\text{ms}}_\alpha(\theta) = \sum_{i=0}^{k-1} \widetilde{\text{ms}}_\alpha(p_i, p_{i+1})$. Also, let $\tilde{A}_\alpha(\theta) = \widetilde{\text{ms}}_\alpha(\theta)/|\theta|$, and define

$$\tilde{A}_\alpha^{\max} = \max\{\tilde{A}_\alpha(\theta) : \theta \text{ is a cyclic sequence of processors}\}.$$

LEMMA 3.7. $A_\alpha^{\max} = \tilde{A}_\alpha^{\max}$.

Proof. Consider any cyclic sequence of processors $\theta = p_0, \ldots, p_k$ (where $p_k = p_0$), and sum the estimates around the cycle as follows:

$$\sum_{i=0}^{k-1} \tilde{\mathrm{ms}}_\alpha(p_i, p_{i+1}) = \sum_{i=0}^{k-1} [\mathrm{ms}_\alpha(p_i, p_{i+1}) + S_{\alpha,p_i} - S_{\alpha,p_{i+1}}].$$

However, the values for S_{α,p_i} cancel each other and we get $\sum_{i=0}^{k-1} \mathrm{ms}_\alpha(p_i, p_{i+1})$, which is $\mathrm{ms}_\alpha(\theta)$. Since this holds for every cyclic sequence of processors, it also holds for any sequence θ where A_α is maximized, i.e., where $A_\alpha(\theta) = A_\alpha^{\max}$. □

Thus, we have the following function for computing corrections.

FUNCTION SHIFTS

Given inputs $\tilde{\mathrm{ms}}_\alpha(p,q) = \mathrm{ms}_\alpha(p,q) + S_{\alpha,p} - S_{\alpha,q}$ for every pair of processors p and q.

1. Compute $A_\alpha^{\max}$ (by computing $\tilde{A}_\alpha^{\max}$).
2. Select an arbitrary root processor r. The correction for each processor $p \in P$ is $\mathrm{dist}_w(r,p)$—the distance in the (complete) directed graph relative to the weights $w(p,q) = A_\alpha^{\max} - \tilde{\mathrm{ms}}_\alpha(p,q)$.

The value of $\tilde{A}_\alpha^{\max}$ can be computed in step 1 by using an algorithm of Karp [6] that runs in $O(n^3)$ time. By Lemma 3.7, this is equivalent to computing $A_\alpha^{\max}$. By definition, $A_\alpha^{\max} \geq A_\alpha(\theta) = \mathrm{ms}_\alpha(\theta)/|\theta|$ for any cycle θ. Therefore,

$$\sum_{\langle p,q\rangle \in \theta} (A_\alpha^{\max} = \tilde{\mathrm{ms}}_\alpha(p,q)) = |\theta| A_\alpha^{\max} - \tilde{\mathrm{ms}}_\alpha(\theta) \geq 0.$$

This implies that there are no negative weight cycles in the complete graph with the weights $w(p,q) = A_\alpha^{\max} - \tilde{\mathrm{ms}}_\alpha(p,q)$. Thus, the distances can be computed as in step 2.

THEOREM 3.8. *The function* SHIFTS *computes optimal corrections with precision* $A_\alpha^{\max}$ *in each execution* α.

Proof. Denote by $f(\alpha)$ the vector of corrections computed by SHIFTS given $\tilde{\mathrm{ms}}_\alpha(p,q)$. We will show that $\bar{\rho}(\alpha, f) \leq A_\alpha^{\max}$; it follows from Theorem 3.6 that these are optimal corrections.

To prove that $\bar{\rho}(\alpha, f) \leq A_\alpha^{\max}$ we need to show that $\rho(\alpha', f(\alpha')) \leq A_\alpha^{\max}$ for any admissible execution $\alpha' \equiv \alpha$. It suffices to show that for every pair of processors p and q,

$$S_{\alpha',p} - f(\alpha',p) - S_{\alpha',q} + f(\alpha',q) \leq A_\alpha^{\max}.$$

Fix some pair of processors p and q. By the definition of the function SHIFTS, $f(\alpha,q) = \mathrm{dist}_w(r,q)$ and $f(\alpha,p) = \mathrm{dist}_w(r,p)$, relative to the weights $w(p,q) = A_\alpha^{\max} - \tilde{\mathrm{ms}}_\alpha(p,q)$. Thus

$$f(\alpha,q) - f(\alpha,p) = \mathrm{dist}_w(r,q) - \mathrm{dist}_w(r,p) \leq w(p,q) = A_\alpha^{\max} - \tilde{\mathrm{ms}}_\alpha(p,q).$$

Adding $\tilde{\mathrm{ms}}_\alpha(p,q) - \mathrm{ms}_\alpha(p,q)$ to both sides, we get

$$\tilde{\mathrm{ms}}_\alpha(p,q) - \mathrm{ms}_\alpha(p,q) + f(\alpha,q) - f(\alpha,p) \leq A_\alpha^{\max} - \mathrm{ms}_\alpha(p,q).$$

By the definition of estimated maximal shifts, $\tilde{\mathrm{ms}}_\alpha(p,q) - \mathrm{ms}_\alpha(p,q) = S_{\alpha,p} - S_{\alpha,q}$. Hence

$$S_{\alpha,p} - S_{\alpha,q} + f(\alpha,q) - f(\alpha,p) + \mathrm{ms}_\alpha(p,q) \leq A_\alpha^{\max}.$$

Since $\alpha \equiv \alpha'$, Claim 2.1 implies that $f(\alpha) = f(\alpha')$, and thus

$$S_{\alpha,p} - S_{\alpha,q} + f(\alpha',q) - f(\alpha',p) + \mathrm{ms}_\alpha(p,q) \leq A_\alpha^{\max}.$$

By Claim 3.4, $S_{\alpha',p} - S_{\alpha',q} \leq S_{\alpha,p} - S_{\alpha,q} + \mathrm{ms}_\alpha(p,q)$, and thus,

$$S_{\alpha',p} - f(\alpha',p) - S_{\alpha',q} + f(\alpha',q) \leq A_\alpha^{\max},$$

as needed. □

This theorem implies that given estimates $\tilde{\mathrm{ms}}_\alpha$ of the maximal shifts, we can compute optimal corrections.

4. Calculating estimates in local systems. In the previous section, we reduced the problem of designing an optimal clock synchronization algorithm to the problem of finding the estimates $\tilde{\mathrm{ms}}_\alpha(p,q)$ of maximal shifts for each pair of processors p and q. Given such estimates, the clock synchronization problem can be solved by computing the function SHIFTS. Next we show how to calculate $\tilde{\mathrm{ms}}_\alpha$. In this section, we show how to compute these estimates in the natural class of local systems. Intuitively, in local systems the delays of messages sent to a pair of processors, e.g., along edges connecting them, do not depend on the delays of messages sent to other processors.

For local systems, estimates $\tilde{\mathrm{ms}}(p,q)$ can be computed in two steps. In the first step, local (pairwise) estimates $\tilde{\mathrm{mls}}(p,q)$ are computed. In the second step, the desired global estimates $\tilde{\mathrm{ms}}(p,q)$ are produced by combining the local estimates. In this section we deal only with the second step; i.e., we show how to compute global estimates from local estimates. In the next section, we compute the local estimates based on the views for several specific systems.

To design a clock synchronization algorithm for a specific local system, only the calculation of local estimates must be modified. As illustrated by the particular systems discussed in the next section, this calculation handles each pair of processors separately. This significantly simplifies reasoning and allows us to deal with combinations of several assumptions on the same or on different edges, as we show at the end of this section.

4.1. Local systems. Informally, a system is local if its admissible executions can be expressed as the intersection of sets of executions, each set restricting only the views of a specific pair of processors.

DEFINITION 4.1. *A set of executions $\mathcal{A}_{p,q}$ is* local *to p and q provided that for every execution $\alpha \in \mathcal{A}_{p,q}$, if there exists an execution α' such that $\alpha'|q = \alpha|q$ and $\alpha'|p = \alpha|p$, then $\alpha' \in \mathcal{A}_{p,q}$.*

Note that this implies that $\mathcal{A}_{p,q}$ allows arbitrary shifts as long as both p and q are shifted by the same amount, which is reworded as follows.

CLAIM 4.1. *Assume $\mathcal{A}_{p,q}$ is local to p and q. Let $\alpha \in \mathcal{A}_{p,q}$, and let $\vec{s} = \langle s_1, \ldots, s_n \rangle$ be a vector of shifts such that $s_p = s_q$. Then* $\mathrm{shift}(\alpha, \vec{s}) \in \mathcal{A}_{p,q}$.

Let $\alpha \in \mathcal{A}_{p,q}$. We say that s is a *locally admissible shift* of q w.r.t. p in α if there exists a vector of shifts $\vec{s} = \langle s_1, \ldots, s_n \rangle$ such that $s_q - s_p = s$ and $\mathrm{shift}(\alpha, \vec{s}) \in \mathcal{A}_{p,q}$. We have the following claim.

CLAIM 4.2. *Assume $\mathcal{A}_{p,q}$ is local to p and q, and consider an execution $\alpha \in \mathcal{A}_{p,q}$. A value s is a locally admissible shift of q w.r.t. p in α if and only if for every vector of shifts $\vec{s} = \langle s_1, \ldots, s_n \rangle$ with $s_q - s_p = s$,* $\mathrm{shift}(\alpha, \vec{s}) \in \mathcal{A}_{p,q}$.

Define the *maximal local shift of q w.r.t. p in α* to be

$$\mathrm{mls}_\alpha(p,q) = \sup\{s : s \text{ is a locally admissible shift of } q \text{ w.r.t. } p \text{ in } \alpha\}.$$

Intuitively, $\text{mls}_\alpha(p,q)$ is the maximal possible shift of q w.r.t. p in α, when the admissibility of processors other than p and q need not be preserved.

We usually leave the sets $\mathcal{A}_{p,q}$ unspecified when they are clear from the context. Later, when we want to specify different assumptions on the same pair of processors, we consider more than one local set of executions for this pair. In this case, to distinguish between different local sets for processors p and q we will use $\mathcal{A}_{p,q}, \mathcal{A}'_{p,q}, \mathcal{A}''_{p,q}$, etc.

DEFINITION 4.2. *A set of admissible executions $\mathcal{A}$ is* local *if there exist local sets $\mathcal{A}_{p,q}$ such that $\mathcal{A} = \bigcap_{p,q \in P} \mathcal{A}_{p,q}$, and for every p and q, $\mathcal{A}_{p,q}$ is local to p and q.*

Claim 4.2 implies the following.

CLAIM 4.3. *Assume $\mathcal{A}$ is local. Let $\vec{s} = \langle s_1, \ldots, s_n \rangle$ be a vector of shifts and let $\alpha \in \mathcal{A}$. Then* $\text{shift}(\alpha, \vec{s}) \in \mathcal{A}$ *if and only if $s_q - s_p$ is a locally admissible shift of q w.r.t. p in α, for every pair of processors p and q.*

Notice that $\text{mls}_\alpha(p,q) \geq \text{ms}_\alpha(p,q)$ and $\text{mls}_\alpha(p,q) \geq 0$. We say that $\text{mls}_\alpha(p,q)$ is a *local shift*, while $\text{ms}_\alpha(p,q)$ is a *global shift.*

Note that $\text{mls}_\alpha(p,q)$ may differ from $\text{mls}_\alpha(q,p)$. However, if $\text{shift}(\alpha, \langle s_1, \ldots, s_n \rangle)$ is in $\mathcal{A}_{p,q}$, then $s_q - s_p$ is a locally admissible shift of q w.r.t. p and $s_p - s_q$ is a locally admissible shift of p w.r.t. q. Thus, if s is a locally admissible shift of q w.r.t. p in α, then $-s$ is a locally admissible shift of p w.r.t. q in α.

Throughout the rest of the section, we assume that $\mathcal{A}$ is local with respect to some local sets of executions. Hence, the locally admissible shifts are defined. Furthermore, we assume that the locally admissible shifts have the following property.

ASSUMPTION 1. *Let x and y be two numbers such that $x < y$. For every two processors p, q and every $\alpha \in \mathcal{A}$, if x and y are locally admissible shifts of q w.r.t. p in α, then every value $z \in [y, x]$ is a locally admissible shift of q w.r.t. p in α.*

Intuitively, this assumption implies that the possible shifts constitute a continuous interval without any singularity points, and therefore, it holds in most natural applications.

4.2. From local shifts to global shifts. Our goal is to compute global estimates $\widetilde{\text{ms}}_\alpha(p,q)$ from local estimates $\widetilde{\text{mls}}_\alpha(p,q)$. In this section, as a first step in this direction, we show how to obtain maximal global shifts $\text{ms}_\alpha(p,q)$ from maximal local shifts $\text{mls}_\alpha(p,q)$. This also shows how to derive a lower bound on the precision of clock synchronization from a lower bound on the precision of each edge independently.

Let α be an admissible execution. Denote by $\text{dist}_{w'}(p,q)$ the distance from p to q in the graph G relative to the weights $w'(p,q) = \text{mls}_\alpha(p,q)$. We assume that all the distances $\text{dist}_{w'}$ are finite; it is not difficult to generalize the result for the case of infinite distances.

LEMMA 4.4. *For any pair of processors p and q,* $\text{dist}_{w'}(p,q) \leq \text{ms}_\alpha(p,q)$.

Proof. Assume, by way of contradiction, that for some pair of processors p and q, $\text{dist}_{w'}(p,q) > \text{ms}_\alpha(p,q)$. Thus there exists some value s, $\text{ms}_\alpha(p,q) < s < \text{dist}_{w'}(p,q)$. We show that s is a (globally) admissible shift of q w.r.t. p in α, which contradicts the definition of $\text{ms}_\alpha(p,q)$.

Since $s > \text{ms}_\alpha(p,q)$, it follows that $s > 0$. Thus we can write $s = c \cdot \text{dist}_{w'}(p,q)$, for some real number c, $0 < c < 1$. Fix some pair of processors j and k. By Assumption 1, $c \cdot \text{mls}_\alpha(k,j)$ is a locally admissible shift of j w.r.t. k in α. In addition, $c \cdot \text{mls}_\alpha(j,k)$ is a locally admissible shift of k w.r.t. j in α, and thus $-c \cdot \text{mls}_\alpha(j,k)$ is a locally admissible shift of j w.r.t. k in α.

For every processor i define $s_i = c \cdot \text{dist}_{w'}(p,i)$; note that $s_p = 0$ and $s_q = s$. We now show that $s_j - s_k$ is a locally admissible shift of j w.r.t. k in α. By the triangle inequality

$$\mathrm{dist}_{w'}(p,j) \leq \mathrm{dist}_{w'}(p,k) + w'(k,j),$$

and since $w'(k,j) = \mathrm{mls}_\alpha(k,j)$, we have (by changing sides)

$$\mathrm{dist}_{w'}(p,j) - \mathrm{dist}_{w'}(p,k) \leq \mathrm{mls}_\alpha(k,j).$$

Since $c > 0$, by multiplying by c and substituting s_j and s_k, we get

$$s_j - s_k \leq c \cdot \mathrm{mls}_\alpha(k,j).$$

By similar reasoning.

$$s_k - s_j \leq c \cdot \mathrm{mls}_\alpha(j,k).$$

This implies

$$-c \cdot \mathrm{mls}_\alpha(j,k) \leq s_j - s_k \leq c \cdot \mathrm{mls}_\alpha(k,j).$$

Since $-c \cdot \mathrm{mls}_\alpha(j,k)$ and $c \cdot \mathrm{mls}_\alpha(k,j)$ are locally admissible shifts of j w.r.t. k in α, Assumption 1 implies that $s_j - s_k$ is a locally admissible shift of j w.r.t. k in α.

Since this holds for any j and k, Claim 4.3 implies that $\mathrm{shift}(\alpha, \langle s_1, \ldots, s_n\rangle)$ is in $\mathcal{A}$. Therefore, $s = s_q - s_p$ is a (globally) admissible shift of q w.r.t. p in α. Since $s > \mathrm{ms}_\alpha(p,q)$, this contradicts the definition of $\mathrm{ms}_\alpha(p,q)$. □

We now show that $\mathrm{ms}_\alpha(p,q) = \mathrm{dist}_{w'}(p,q)$, by proving the converse inequality.

LEMMA 4.5. *For any two processors p and q,* $\mathrm{ms}_\alpha(p,q) \leq \mathrm{dist}_{w'}(p,q)$.

Proof. Let s be an arbitrary admissible shift of q w.r.t. p in α. Then there exists a vector of shifts $\vec{s} = \langle s_1, \ldots, s_n\rangle$ with $s = s_q - s_p$, such that shift $(\alpha, \vec{s}) \in \mathcal{A}$. Consider a shortest path $p_0, \ldots, p_k$ from $p = p_0$ to $q = p_k$ w.r.t. the weights w'. Summing over the path we get

$$(1) \qquad \sum_{i=1}^{k}(s_{p_i} - s_{p_{i-1}}) = s_{p_k} - s_{p_0} = s_q - s_p = s.$$

Since $\mathrm{shift}(\alpha, \vec{s}) \in \mathcal{A}$, Claim 4.3 implies that $s_{p_i} - s_{p_{i-1}}$ is a locally admissible shift of p_i w.r.t. p_{i-1} in α. Hence,

$$(2) \qquad s_{p_i} - s_{p_{i-1}} \leq \mathrm{mls}_\alpha(p_{i-1}, p_i) = w'(p_{i-1}, p_i)$$

for each $i - 1, \ldots, k$. By combining (1) and (2) we get

$$s = \sum_{i=1}^{k}(s_{p_i} - s_{p_{i-1}}) \leq \sum_{i=1}^{k} w'(p_{i-1}, p_i) = \mathrm{dist}_{w'}(p,q).$$

Therefore, $s \leq \mathrm{dist}_{w'}(p,q)$. Since this holds for any admissible shift, it follows that $\mathrm{ms}_\alpha(p,q) \leq \mathrm{dist}_{w'}(p,q)$. □

THEOREM 4.6. *For any admissible execution α and any two processors p and q, $\mathrm{ms}_\alpha(p,q)$ can be computed from* $\mathrm{mls}_\alpha(p,q)$.

Proof. For any admissible execution α and any two processors p and q, Lemma 4.4 implies that $\mathrm{ms}_\alpha(p,q) \geq \mathrm{dist}_{w'}(p,q)$, where $w'(p,q) = \mathrm{mls}_\alpha(p,q)$. Lemma 4.5 implies that $\mathrm{ms}_\alpha(p,q) = \mathrm{dist}_{w'}(p,q)$. The claim follows since $\mathrm{dist}_{w'}(p,q)$ depends only on $\mathrm{mls}_\alpha(p,q)$. □

4.3. Using estimates for local shifts. Now the issue is how to compute the values $\tilde{\mathrm{ms}}_\alpha(p,q)$ that are needed as inputs to the function SHIFTS. We assume that the function is provided with estimates of the local shifts. Under this assumption the computation can be accomplished by the following function.

FUNCTION GLOBAL ESTIMATES

Given inputs $\tilde{\mathrm{mls}}_\alpha(p,q) = \mathrm{mls}_\alpha(p,q) + S_{\alpha,p} - S_{\alpha,q}$ for every pair of processors p and q.

1. Compute $\tilde{\mathrm{ms}}_\alpha(p,q)$ by a shortest path computation in G with weights $\tilde{\mathrm{mls}}_\alpha(p,q)$.

THEOREM 4.7. *The function* GLOBAL ESTIMATES *computes* $\tilde{\mathrm{ms}}_\alpha(p,q)$, *for every pair of processors* p *and* q.

Proof. Observe that the weight of any cycle w.r.t. the weights $\tilde{\mathrm{mls}}_\alpha$ is equal to the weight of the cycle w.r.t. the weights mls_α because the S components cancel. It follows that there are no negative weight cycles in G w.r.t. the weights $\tilde{\mathrm{mls}}_\alpha$. Also, the weight of any path from p to q w.r.t. weights $\tilde{\mathrm{mls}}_\alpha$ is equal to the weight of the path w.r.t. mls_α plus $S_{\alpha,p} - S_{\alpha,q}$. The claim follows from Theorem 4.6. □

By composing functions GLOBAL ESTIMATES and SHIFTS, we can compute the optimal corrections and their precision given only the estimates to the maximal local shifts $\tilde{\mathrm{mls}}_\alpha$. This follows immediately from Theorem 4.7 together with Theorem 3.8.

4.4. A composition theorem. In many systems, several constraints are imposed on the delay of messages. For example, it is possible that there is a bound on the delay in each direction of the link as well as a bound on the difference in message delay in opposite directions. In these cases, the system is local w.r.t. several sets of executions, each of which is local to the same pair of processors p and q. We now show how to combine several sets of executions local to p and q into a single complex set of executions local to p and q. This allows us to deal with local systems by regarding each pair of processors and each assumption separately; this will be useful in the next section.

We remark that the theory developed so far already allows us to deal with local systems where different pairs of processors obey different types of constraints.

Note that the notion of an admissible shift (and the derived notion of maximal shift) is defined in the context of a specific set of admissible executions. To develop the results in this section it is convenient to state this fact explicitly by saying that a value is an admissible shift (or maximal shift) *under* $\mathcal{A}$, where $\mathcal{A}$ is some set of executions.

For some pair of processors p and q, let $\mathcal{A}'_{p,q}$ be a set of executions local to p and q, and let $\mathcal{A}''_{p,q}$ be another set of executions local to p and q. Denote $\mathcal{A}_{p,q} = \mathcal{A}'_{p,q} \cap \mathcal{A}''_{p,q}$. It is easy to see that $\mathcal{A}_{p,q}$ is local to p and q. For any execution $\alpha \in \mathcal{A}$, let $\mathrm{mls}'_\alpha(p,q)$ be the maximal local shift of q w.r.t. p in α under $\mathcal{A}'_{p,q}$. Similarly, define $\mathrm{mls}_\alpha(p,q)$ and $\mathrm{mls}''_\alpha(p,q)$. We have the following theorem.

THEOREM 4.8. $\mathrm{mls}_\alpha(p,q) = \min\{\mathrm{mls}'_\alpha(p,q), \mathrm{mls}''_\alpha(p,q)\}$.

Proof. The fact that $\mathrm{mls}_\alpha(p,q) \le \mathrm{mls}'_\alpha(p,q)$ follows immediately since every execution in $\mathcal{A}_{p,q}$ is an execution in $\mathcal{A}'_{p,q}$. Thus if s is a locally admissible shift of q w.r.t. p in α under $\mathcal{A}_{p,q}$, then it is a locally admissible shift of q w.r.t. p in α under $\mathcal{A}'_{p,q}$. Similarly $\mathrm{mls}_\alpha(p,q) \le \mathrm{mls}''_\alpha(p,q)$. Therefore, $\mathrm{mls}_\alpha(p,q) \le \min\{\mathrm{mls}'_\alpha(p,q), \mathrm{mls}''_\alpha(p,q)\}$.

Assume for contradiction that s is a value such that $\mathrm{mls}_\alpha(p,q) < s < \min\{\mathrm{mls}'_\alpha(p,q), \mathrm{mls}''_\alpha(p,q)\}$. Since $\mathrm{mls}_\alpha(p,q) < s < \mathrm{mls}'_\alpha(p,q)$, Assumption 1 implies that s is a locally admissible shift of q w.r.t. p in α under $\mathcal{A}'_{p,q}$. Similarly, s

is also a locally admissible shift of q w.r.t. p in α under $\mathcal{A}''_{p,q}$. By Claim 4.2, for every vector of shifts $\vec{s} = \langle s_1, \ldots, s_n \rangle$, such that $s = s_q - s_p$, $\text{shift}(\alpha, \vec{s})$ is in both $\mathcal{A}'_{p,q}$ and $\mathcal{A}''_{p,q}$. Therefore, $\text{shift}(\alpha, \vec{s})$ is in $\mathcal{A}_{p,q}$. Thus, s is a locally admissible shift of q w.r.t. p in α under $\mathcal{A}_{p,q}$, a contradiction since $\text{mls}_\alpha(p,q) < s$. Therefore $\text{mls}_\alpha(p,q) = \min\{\text{mls}'_\alpha(p,q), \text{mls}''_\alpha(p,q)\}$. □

5. Clock synchronization for specific delay assumptions. We now show how to compute estimated maximal local shifts for specific sets of executions $\mathcal{A}_{p,q}$ local to p and q, given the views. By Theorems 4.7 and 4.8, this implies a clock synchronization algorithm that computes optimal corrections for any system whose set of admissible executions is the intersection of any collection of sets of these types. By Theorem 4.7, all we must show is how to compute the estimates of the maximal local shifts $\widetilde{\text{mls}}_\alpha(p,q)$. This calculation is based on estimates for the delays (defined below), which can easily be computed from the views of the processors.

The *estimated delay* $\tilde{\text{d}}(m)$ of a message m sent from p to q is the actual (real-time) delay plus the difference in (real-time) start times of the processors; that is, $\tilde{\text{d}}(m) = \text{d}(m) + S_{\alpha,p} - S_{\alpha,q}$. This is similar to the definitions of estimated maximal global shifts and estimated maximal local shifts. The next lemma shows that the estimated delay can be computed from the views.

LEMMA 5.1. *Given the views of processors p and q in an execution α, it is possible to compute the estimated delay $\tilde{\text{d}}(m)$ of any message m sent from p to q.*

Proof. Let $t_p(m)$ denote the local (p's) clock time when p sent the message m according to p's view; similarly, $t_q(m)$ denotes the local (q's) clock time when q received the message m according to q's view. By property 4 of histories as defined in §2.2, m was sent at real time $t_p(m) + S_{\alpha,p}$; similarly, m was received at real time $t_q(m) + S_{\alpha,q}$. It follows that the delay of m is $\text{d}(m) = (t_q(m) + S_{\alpha,q}) - (t_p(m) + S_{\alpha,p})$. Hence, $\tilde{\text{d}}(m) = \text{d}(m) + S_{\alpha,p} - S_{\alpha,q} = t_q(m) - t_p(m)$. Since messages are unique, $t_p(m)$ and $t_q(m)$ can be computed from the views of p and q. □

5.1. Bounds on the delay. In the systems considered in [4, 11], there is an upper and a lower bound on the transmission delay for any edge. We extend this assumption by allowing edges without upper bounds, in which case we say that the upper bound is ∞. In particular, this gives optimal clock synchronization for completely asynchronous networks where there are no bounds on the delay.

Consider a set $\mathcal{A}_{p,q}[l,u]$ where l and u give bounds (real numbers) for each ordered pair of processors p and q, such that $0 \le l(p,q) \le u(p,q) \le \infty$. Execution α is in $\mathcal{A}_{p,q}[l,u]$ if the delay of every message sent from p to q is in the range $[l(p,q), u(p,q)]$ and the delay of every message from q to p is in the range $[l(q,p), u(q,p)]$. Clearly, $\mathcal{A}_{p,q}[l,u]$ is local to p and q.

The maximal delay of a message received by q from p in execution α is denoted $\text{d}^{\max}_\alpha(p,q)$. Similarly, the minimal delay of a message received by q from p in α is denoted $\text{d}^{\max}_\alpha(p,q)$. If no message was received by q from p in α, then $\text{d}^{\max}_\alpha(p,q) = -\infty$ and $\text{d}^{\min}_\alpha(p,q) = \infty$. We first observe that in such systems, $\text{mls}_\alpha(p,q)$ depends only on the maximal and minimal delays between p and q.

LEMMA 5.2. *Let α be an execution of $(P, \mathcal{A}_{p,q}[l,u])$. Then*

$$\text{mls}_\alpha(p,q) = \min\{(u(q,p) - \text{d}^{\max}_\alpha(q,p)), (\text{d}^{\min}_\alpha(p,q) - l(p,q))\}.$$

Proof. We can partition the constraints on the communication between p and q in two: the conditions on the delay of messages from p to q, and the conditions on the delay of messages from q to p. This is done by expressing the set $\mathcal{A}_{p,q}[l,u]$ as the intersection of $\mathcal{A}_{\langle q,p\rangle}[l,u]$, which constrains the messages from q to p, and $\mathcal{A}_{\langle p,q\rangle}[l,u]$, which constrains the messages from p to q. Let $\mathrm{mls}'_\alpha(p,q)$ be the maximal local shift of q w.r.t. p in α under $\mathcal{A}_{\langle q,p\rangle}[l,u]$; $\mathrm{mls}''_\alpha(p,q)$ is defined similarly under $\mathcal{A}_{\langle p,q\rangle}[l,u]$.

We first show that $\mathrm{mls}'_\alpha(p,q) = u(q,p) - \mathrm{d}^{\max}_\alpha(q,p)$. It is obvious that $\mathrm{mls}'_\alpha(p,q) \leq u(q,p) - \mathrm{d}^{\max}_\alpha(q,p)$. Assume, by way of contradiction, that $s > u(q,p) - \mathrm{d}^{\max}_\alpha(q,p)$ is a locally admissible shift of q w.r.t. p in α. This immediately implies that $\mathrm{d}^{\max}_\alpha(q,p) > -\infty$; i.e., at least one message was received by p from q.

Denote $\alpha' = \mathrm{shift}(\alpha, \langle s_1, \ldots, s_n\rangle)$, where $s_q = s$ and $s_i = 0$ for all $i \neq q$. By Claim 4.2, α' is in $\mathcal{A}_{\langle q,p\rangle}$. By Claim 3.2, if a message m from q to p has delay d in α, then m has delay $\mathrm{d}+s$ in α'. Thus, $\mathrm{d}^{\max}_{\alpha'}(q,p) = \mathrm{d}^{\max}_\alpha(q,p) - s$. Since $s > u(q,p) - \mathrm{d}^{\max}_\alpha(q,p)$ and $\mathrm{d}^{\max}_\alpha(q,p) > -\infty$, it follows that the maximal delay of a message from q to p in α' is strictly greater than $u(q,p)$, which is a contradiction.

In a similar manner, we show that $\mathrm{mls}''_\alpha(p,q) = \mathrm{d}^{\min}_\alpha(p,q) - l(p,q)$. It is obvious that $\mathrm{mls}''_\alpha(p,q) \leq \mathrm{d}^{\min}_\alpha(p,q) - l(p,q)$. Assume, by way of contradiction, that $s > \mathrm{d}^{\min}_\alpha(p,q) - l(p,q)$ is a locally admissible shift of q w.r.t. p in α. This immediately implies that $\mathrm{d}^{\min}_\alpha(p,q) < \infty$; i.e., at least one message was received by q from p.

Let α' be as defined above. By Claim 4.2, α' is in $\mathcal{A}_{\langle p,q\rangle}$. By Claim 3.2, if a message m from p to q has delay d in α, then m has delay $\mathrm{d} - s$ in α'. Thus, $\mathrm{d}^{\min}_{\alpha'}(p,q) = \mathrm{d}^{\min}_\alpha(p,q) - s$. Since $s > \mathrm{d}^{\min}_\alpha(p,q) - l(p,q)$ and $\mathrm{d}^{\min}_\alpha(p,q) < \infty$, it follows that the minimal delay of a message from p to q in α' is less than $l(p,q)$, which is a contradiction.

The claim now follows from Theorem 4.8. □

Lemma 5.2 gives the maximal local shifts as a function of the actual maximal and minimal delays. However, the views of the processors give only estimates of the delays, not the delays themselves. Yet, the estimates of the delays give an estimate for the maximal local shift $\widetilde{\mathrm{mls}}_\alpha(p,q)$. Formally, the *estimated maximal delay* is defined as

$$\tilde{\mathrm{d}}^{\max}_\alpha(p,q) = \mathrm{d}^{\max}_\alpha(p,q) + S_{\alpha,p} - S_{\alpha,q},$$

while the *estimated minimal delay* is defined as

$$\tilde{\mathrm{d}}^{\min}_\alpha(p,q) = \mathrm{d}^{\min}_\alpha(p,q) + S_{\alpha,p} - S_{\alpha,q}.$$

We have the following result.

COROLLARY 5.3. *Let α be an admissible execution of $(P, \mathcal{A}_{p,q}[l,u])$. Then*

$$\widetilde{\mathrm{mls}}_\alpha(p,q) = \min\{(u(q,p) - \tilde{\mathrm{d}}^{\max}_\alpha(q,p)), (\tilde{\mathrm{d}}^{\min}_\alpha(p,q) - l(p,q))\}.$$

Note that $\tilde{\mathrm{d}}^{\max}$ and $\tilde{\mathrm{d}}^{\min}$ can be computed from the views of p and q, since $\tilde{\mathrm{d}}^{\min}_\alpha(p,q)$ is the minimum of $\tilde{\mathrm{d}}(m)$ for all messages m received by q from p in α and $\tilde{\mathrm{d}}^{\max}_\alpha(p,q)$ is the maximum of $\tilde{\mathrm{d}}(m)$ for all messages m received by q from p in α.

If we make the natural assumption that all delays are nonnegative, we get a general bound on mls and $\widetilde{\mathrm{mls}}$ (without any other bounds on the delay).

COROLLARY 5.4. *Let α be an admissible execution of a system $(P, \mathcal{A}_{p,q})$ local to p, q. Then $\mathrm{mls}_\alpha(p,q) \leq \mathrm{d}^{\min}_\alpha(p,q)$ and $\widetilde{\mathrm{mls}}_\alpha(p,q) \leq \tilde{\mathrm{d}}^{\min}_\alpha(p,q)$.*

5.2. Links with bounds on the round-trip delay bias. In many communication links there are no tight bounds on the transmission delays. However, whenever the traffic load in one direction of a link is high, the load in the opposite direction of the link is also high. Thus, it is possible to give a bound on the difference, or bias, between the delay in one direction and the delay in the opposite direction. For the purpose of illustrating our techniques, we simplify the assumption and require that the difference between the delay of any pair of messages in opposite direction be bounded. We now show how to calculate maximal local shifts for links in this case. It is possible to generalize our results to the more realistic scenario in which this assumption holds only for messages that were sent "around the same time."

Specifically, we associate a nonnegative number $b(p,q)$ with processors p,q. An execution α is in $\mathcal{A}_{p,q}[b]$ if for any message m received by p from q and any message m' received by q from p, $|\mathrm{d}_\alpha(m) - \mathrm{d}_\alpha(m')| \leq b(p,q)$. We also restrict $\mathcal{A}_{p,q}[b]$ to nonnegative delays; i.e., for every message m, $\mathrm{d}(m) \geq 0$. The next lemma shows that $\mathrm{mls}_\alpha(p,q)$ depends only on the maximal and minimal delays between p and q.

LEMMA 5.5. *Let α be an admissible execution of $(P, \mathcal{A}_{p,q}[b])$. Then*

$$\mathrm{mls}_\alpha(p,q) = \min\left\{\mathrm{d}_\alpha^{\min}(p,q), \frac{1}{2}[b(p,q) + \mathrm{d}_\alpha^{\min}(p,q) - \mathrm{d}_\alpha^{\max}(q,p)]\right\}.$$

Proof. Consider the following two sets of admissible executions local to p and q. The first set $\mathcal{A}'_{p,q}$ contains every execution α such that the delay of every message in α is nonnegative; denote the maximal local shifts under $\mathcal{A}'_{p,q}$ by mls'_α. The second set $\mathcal{A}''_{p,q}$ is like $\mathcal{A}_{p,q}[b]$ except that the delays are allowed to be negative; denote the maximal local shifts under $\mathcal{A}''_{p,q}$ by mls''_α. Clearly $\mathcal{A}_{p,q}[b]$ is the intersection of $\mathcal{A}'_{p,q}$ and $\mathcal{A}''_{p,q}$. By Theorem 4.8, $\mathrm{mls}_\alpha(p,q) = \min\{\mathrm{mls}'_\alpha(p,q), \mathrm{mls}''_\alpha(p,q)\}$. By Lemma 5.2, $\mathrm{mls}'_\alpha(p,q) = \mathrm{d}_\alpha^{\min}(p,q)$. Therefore, it suffices to prove that $\mathrm{mls}''_\alpha(p,q) = \frac{1}{2}[b(p,q) + \mathrm{d}_\alpha^{\min}(p,q) - \mathrm{d}_\alpha^{\max}(q,p)]$.

Fix a value s and denote $\alpha' = \mathrm{shift}(\alpha, \langle s_1, \ldots, s_n\rangle)$, where $s_q = s$ and $s_i = 0$ for all $i \neq q$. By Claim 4.2, s is a locally admissible shift of q w.r.t. p in α if and only if $\alpha' \in \mathcal{A}_{p,q}(\epsilon)$.

By Claim 3.2, for any message m received by p from q, $\mathrm{d}_{\alpha'}(m) = \mathrm{d}_\alpha(m) + s$. Similarly, for any message m' received by q from p, $\mathrm{d}_{\alpha'}(m') = \mathrm{d}_\alpha(m') - s$. Therefore,

$$\mathrm{d}_{\alpha'}(m') - \mathrm{d}_{\alpha'}(m) = \mathrm{d}_\alpha(m') - \mathrm{d}_\alpha(m) - 2s,$$

$$\mathrm{d}_{\alpha'}(m) - \mathrm{d}_{\alpha'}(m') = \mathrm{d}_\alpha(m) - \mathrm{d}_\alpha(m') + 2s.$$

The round-trip delay bias of an arbitrary pair of messages m and m' in α' is at most $b(p,q)$ if and only if

$$\mathrm{d}_\alpha(m') - \mathrm{d}_\alpha(m) - 2s \leq b(p,q),$$

$$\mathrm{d}_\alpha(m) - \mathrm{d}_\alpha(m') + 2s \leq b(p,q).$$

Since α is admissible and $s \geq 0$, the first inequality trivially holds. Hence, s is a locally admissible shift of q w.r.t. p if and only if

$$s \leq \frac{1}{2}[b(p,q) + \mathrm{d}_\alpha(m') - \mathrm{d}_\alpha(m)].$$

Namely,

$$s \leq \frac{1}{2}[b(p,q) + \mathrm{d}_\alpha^{\min}(p,q) - \mathrm{d}_\alpha^{\max}(q,p)].$$

Thus $\text{mls}''_\alpha(p,q) = \frac{1}{2}[b(p,q) + \text{d}^{\min}_\alpha(p,q) - \text{d}^{\max}_\alpha(q,p)]$. □

COROLLARY 5.6. *Let α be an admissible execution of $(P, \mathcal{A}[b])$. Then*

$$\widetilde{\text{mls}}_\alpha(p,q) = \min\left\{\tilde{\text{d}}^{\min}_\alpha(p,q), \frac{1}{2}[b(p,q) + \tilde{\text{d}}^{\min}_\alpha(p,q) - \tilde{\text{d}}^{\max}_\alpha(q,p)]\right\}.$$

5.3. Multicast networks. Communication in many networks is performed through broadcast media where a message is transmitted simultaneously to a subset of the processors. Multicast transmission may often have useful timing properties for clock synchronization. In this subsection we investigate a simple timing property: there exists a bound ε on the difference between the arrival times of a message at different processors. We do not assume that there is any bound on the delay of any individual message.

Optimal clock synchronization algorithms for multicast networks that have a bound on the differences in delay for different processors are presented in [5, 18]. Our solution demonstrates the usefulness of the reductions of the preceding sections. To provide optimal clock synchronization under the multicast assumption we need only to find a way of defining local shifts. This, somewhat surprisingly, turns out to be an easy task. Furthermore, the broadcast model can be limited to specific subsets corresponding to subnetworks of an internet and combined with the other assumptions using the composition theorem.

To define this assumption, we allow events of the form $\text{send}(p, m, Q)$ for all messages m and sets of processors Q; this event represents a multicast of m to the processors in Q. The definition of an execution is modified so that there is a one-to-one correspondence between the messages received by p from k to messages sent by k to p or multicast by k to a set Q that includes p for any processors p and k. Let $\text{d}(p,m)$ denote the difference between the time the message m is multicast by some processor k and the time processor p receives it; this is the *delay of the message m to p*. The *estimated delay of the message m to p* in α is $\tilde{\text{d}}(p,m) = \text{d}(p,m) + S_{\alpha,k} - S_{\alpha,p}$. As before, the estimated delay of a message from k to p can be computed from the views of p and k.

The system is the pair $(P, \mathcal{A}(\varepsilon))$, where $\mathcal{A}(\varepsilon)$ is the intersection of local sets $\mathcal{A}_{p,q}(\varepsilon)$ for every unordered pair of processors p and q. An execution α is in $\mathcal{A}_{p,q}(\varepsilon)$ if for every message m multicast to both p and q, $|\text{d}(p,m) - \text{d}(q,m)| \leq \varepsilon$. That is, m reaches p at most ε time units after it reaches q, and vice versa.

Note that $\mathcal{A}_{p,q}(\varepsilon)$ is local to p and q. This is because any shift applied to both p and q does not change the difference $\text{d}(p,m) - \text{d}(q,m)$. The next lemma shows how to calculate $\text{mls}_\alpha(p,q)$.

LEMMA 5.7. *Let α be an admissible execution of $(P, \mathcal{A}_{p,q}(\varepsilon))$. Then*

$$\text{mls}_\alpha(p,q) = \varepsilon + \min_m\{\text{d}(q,m) - \text{d}(p,m)\}.$$

Proof. Fix a value s and denote $\alpha' = \text{shift}(\alpha, \langle s_1, \ldots, s_n\rangle)$, where $s_q = s$ and $s_i = 0$ for all $i \neq q$. By Claim 4.2, s is a locally admissible shift of q w.r.t. p in α if and only if $\alpha' \in \mathcal{A}_{p,q}(\varepsilon)$.

For every message m multicast to both p and q, let $\text{d}'(q,m)$ and $\text{d}'(p,m)$ denote the delay of m in α' for processors p and q, respectively. The value s is a locally admissible shift of q w.r.t. p in α if and only if

$$|\text{d}'(p,m) - \text{d}'(q,m)| \leq \varepsilon$$

for every message m multicast to both p and q. If q is not the sender of m, then $\mathrm{d}'(p,m) = \mathrm{d}(p,m)$ and $\mathrm{d}'(q,m) = \mathrm{d}(q,m) - s$; if q is the sender of m, then $\mathrm{d}'(q,m) = \mathrm{d}(q,m)$ and $\mathrm{d}'(p,m) = \mathrm{d}(p,m) + s$. In both cases, s is a locally admissible shift of q w.r.t. p in α if and only if

$$|\mathrm{d}(p,m) - \mathrm{d}(q,m) + s| \leq \varepsilon$$

for every message m multicast to both p and q.

Hence, s is a locally admissible shift of q w.r.t. p in α if and only if

$$|s| \leq \varepsilon + |\mathrm{d}(p,m) - \mathrm{d}(q,m)|$$

for every message m multicast to both p and q. Since $\mathrm{mls}_\alpha(p,q) \geq 0$, this implies that

$$\mathrm{mls}_\alpha(p,q) = \varepsilon + \min_m\{\mathrm{d}(p,m) - \mathrm{d}(q,m)\},$$

as needed. □

As before, this result also applies to the estimated delays that can be computed from the views.

COROLLARY 5.8. *Let α be an admissible execution of $(P, \mathcal{A}_{p,q}(\varepsilon))$. Then*

$$\widetilde{\mathrm{mls}}_\alpha(p,q) = \varepsilon + \min_m\{\tilde{\mathrm{d}}(p,m) - \tilde{\mathrm{d}}(p,m)\}.$$

Proof. Fix a message m and let k be the sender of m. We have

$$\begin{aligned}\widetilde{\mathrm{mls}}_\alpha(p,q) &= \mathrm{mls}_\alpha(p,q) + S_{\alpha,p} - S_{\alpha,q} \quad \text{(by definition)}\\ &= \varepsilon + \min_m\{\mathrm{d}(q,m) - \mathrm{d}(p,m)\} + S_{\alpha,p} - S_{\alpha,q} \quad \text{(by Lemma 5.7)}\\ &= \varepsilon + \min_m\{(\mathrm{d}(q,m) + S_{\alpha,k} - S_{\alpha,q}) - (\mathrm{d}(p,m) + S_{\alpha,k} - S_{\alpha,p})\}\\ &= \varepsilon + \min_m\{\tilde{\mathrm{d}}(q,m) - \tilde{\mathrm{d}}(p,m)\} \quad \text{(by definition),}\end{aligned}$$

as needed. □

6. Discussion. We have presented a framework for designing optimal clock synchronization algorithms under a variety of assumptions on message delay uncertainty. The general result yields optimal clock synchronization algorithms under the following assumptions: upper and lower bounds on delays are known (including degenerated cases); only a bound on the difference of the round-trip delays is known; and a multicast assumption that bounds the difference in delay in reaching different processors is known. Moreover, the results apply to cases where different links satisfy different assumptions or where the same link satisfies several assumptions. This work extends results of Halpern, Megiddo, and Munshi [4] and introduces a new notion of optimality on any specific instance.

The specific delay assumptions analyzed here are typical of realistic systems, and it seems relatively easy to perform similar analysis for additional delay assumptions. It is our belief that this will lead to the design of optimal clock synchronization algorithms for other message delay assumptions.

In this paper, we only address the issue of computing optimal corrections, given the views of the processors. An interesting open question is to compute the optimal corrections in a distributed manner. To understand the difficulty involved in the distributed implementation of this computation, consider the following straightforward approach. Each pair of neighboring processors p and q compute $\widetilde{\mathrm{mls}}_\alpha(p,q)$ and $\widetilde{\mathrm{mls}}_\alpha(q,p)$ using the estimated delays (which can be deduced from their views). All processors send the estimated maximum local shifts to a distinguished processor (leader). The leader computes the estimated maximum global shifts using function GLOBAL ESTIMATES and a correction value for each processor according to function SHIFTS. Finally, the leader sends the corrections to the processors. Note, however, that the precision obtained by this centralized clock synchronization algorithm is optimal only with respect to the part of the execution that does not include the messages to and from the leader. That is, any additional communication, required for exchanging the views, is bound to change the views themselves. A solution may require the definition of optimality to be relaxed.

Extensions of our work to the truly distributed case appear in [17]. This work also generalizes some of our results to clocks that drift. Other follow-up work includes an investigation of the problem from the knowledge theoretic point of view [14]. Some techniques for developing clock synchronization algorithms for broadcast networks appeared in [3].

Another important open question is to achieve optimal clock synchronization in systems where the probabilistic properties of the message delay distribution are known. This assumption is at the heart of most practical algorithms for clock synchronization [1, 13]. We believe the setting developed here allows one to address this assumption and that this will lead to improvements to these important algorithms.

Finally, an obvious open problem is to make our results to be fault tolerant, following the many works addressing fault-tolerant clock synchronization.

Acknowledgments. We thank Joe Halpern, Marios Mavronicolas, Boaz Patt-Shamir, and the anonymous referees for helpful comments.

REFERENCES

[1] F. CRISTIAN, *Probabilistic clock synchronization*, Distrib. Comput., 3 (1989), pp. 146–158.

[2] D. DOLEV, J. HALPERN, AND H. R. STRONG, *On the possibility and impossibility of achieving clock synchronization*, J. Comput. System Sci., 32 (1986), pp. 230–250.

[3] D. DOLEV, R. REISCHUK, AND H. R. STRONG, *Observable clock synchronization*, in Proc. 13th ACM Symposium on Principles of Distributed Computing, August 1994, Association for Computing Machinery, New York, 1994, pp. 284–293.

[4] J. HALPERN, N. MEGIDDO, AND A. A. MUNSHI, *Optimal precision in the presence of uncertainty*, J. Complexity, 1 (1985), pp. 170–196.

[5] J. HALPERN AND I. SUZUKI, *Clock synchronization and the power of broadcasting*, in Proc. 28th Annual Allerton Conference on Communication, Control, and Computing, Allerton, IL, October 1990, pp. 588–597.

[6] R. M. KARP, *A characterization of the minimum cycle mean in a digraph*, Discrete Math., 23 (1978), pp. 309–311.

[7] H. KOPETZ AND W. OCHSENREITER, *Clock synchronization in distributed real-time systems*, IEEE Trans. Comput., 36 (1987), pp. 933–939.

[8] L. LAMPORT, *Time, clocks and the ordering of events in distributed systems*, Comm. Assoc. Comput. Mach., 21 (1978), pp. 558–565.

[9] L. LAMPORT AND P. MELLIAR-SMITH, *Synchronizing clocks in the presence of faults*, J. Assoc. Comput. Mach., 32 (1985), pp. 52–78.

[10] B. LISKOV, *Practical uses of synchronized clocks in distributed systems*, invited talk at the 9th ACM Symposium on Principles of Distributed Computing, Association for Computing Machinery, New York, 1990; Distrib. Comput., 6 (1993), pp. 211–219.

[11] J. LUNDELIUS AND N. LYNCH, *An upper and lower bound for clock synchronization*, Inform. and Control, 62 (1984), pp. 190–204.

[12] K. Marzullo, *Loosely-coupled distributed services: A distributed time service*, Ph.D. thesis, Stanford University, Stanford, CA, 1983.

[13] D. Mills, *Network time protocol (version 2) specification and implementation*, IEEE Trans. Comm., 39 (1991), pp. 1482–1493.

[14] Y. Moses and B. Bloom, *Knowledge, timed precedence and clocks*, in Proc. 13th ACM Symposium on Principles of Distributed Computing, Los Angeles, August 1994, Association for Computing Machinery, New York, 1994, pp. 294–303.

[15] Y. Ofek, *Generating a fault-tolerant global clock using high-speed control signals for the MetaNet architecture*, IEEE Trans. Comm., 42 (1994), pp. 2179–2188.

[16] Open Software Foundation, *Introduction to OSF DCE*, Open Software Foundation (OSF), Cambridge, MA, December 1991.

[17] B. Patt-Shamir and S. Rajsbaum, *A theory of clock synchronization*, in Proc. 26th ACM Sumposium on Theory of Computing, Association for Computing Machinery, New York, 1994, pp. 810–819.

[18] K. Sugihara and I. Suzuki, *Nearly optimal clock synchronization under unbounded message transmission time*, in Proc. 1988 International Conference on Parallel Processing III, St. Charles, IL, 1988, pp. 14–17.

[19] B. Simons, J. L. Welch, and N. Lynch, *An Overview of Clock Synchronization*, IBM Technical Report RJ 6505, IBM, October 1988.

[20] T. Srikanth and S. Toueg, *Optimal clock synchronization*, J. Assoc. Comput. Mach., 34 (1987), pp. 626–645.

[21] J. L. Welch and N. Lynch, *A new fault tolerant algorithm for clock synchronization*, Inform. and Comput., 77 (1988), pp. 1–36.

SIAM J. COMPUT.
Vol. 25, No. 2, pp. 390–403, April 1996

010

LINEAR-TIME REPRESENTATION ALGORITHMS FOR PROPER CIRCULAR-ARC GRAPHS AND PROPER INTERVAL GRAPHS*

XIAOTIE DENG†, PAVOL HELL‡, AND JING HUANG§

Abstract. Our main result is a linear-time (that is, time $O(m+n)$) algorithm to recognize and represent proper circular-arc graphs. The best previous algorithm, due to A. Tucker, has time complexity $O(n^2)$. We take advantage of the fact that (among connected graphs) proper circular-arc graphs are precisely the graphs orientable as local tournaments, and we use a new characterization of local tournaments. The algorithm depends on repeated representation of portions of the input graph as proper interval graphs. Thus we also find it useful to give a new linear-time algorithm to represent proper interval graphs. This latter algorithm also depends on an orientation characterization of proper interval graphs. It is conceptually simple and does not use complex data structures. As a byproduct of the correctness proof of the algorithm, we also obtain a new proof of a characterization of proper interval graphs by forbidden subgraphs.

Key words. proper circular-arc graphs, proper interval graphs, local tournaments, representation algorithms

AMS subject classifications. 05C85, 05C75, 68Q25

1. Introduction.

An *interval graph* is the intersection graph of a family of linear intervals; a *proper interval graph* is the intersection graph of an inclusion-free family of linear intervals. A *circular-arc graph* is the intersection graph of a family of circular arcs; as above, it is called *proper* if the family can be chosen to be inclusion free. The families of intervals or of circular arcs are called *representations* of the corresponding graphs.

Algorithmic aspects of interval graphs have been extensively studied; cf. [7]. In particular, there are linear-time (i.e., time $O(m+n)$, where m and n are, respectively, the numbers of edges and of vertices of the input graph) algorithms to find a representation of a given input graph by a family of intervals if one exists [5, 14]. It is a long-standing open problem to find a linear-time algorithm for the representation of circular-arc graphs. Here we give a linear-time algorithm for the representation of *proper* circular-arc graphs. Tucker [18] gave a matrix characterization of proper circular-arc graphs and an associated representation algorithm, which runs in time $O(n^2)$ (cf. [15]).

Our algorithm depends on a linear-time method to represent proper interval graphs. Such a method can be extracted from existing algorithms for interval graphs [7, p. 195]. However, we also include in this paper a new linear-time algorithm for the representation of proper interval graphs. This is a simple incremental algorithm which does not use special data structures such as PQ-trees [5, 14]. Moreover, it is formulated in a way that makes it convenient to use as a subroutine for our main algorithm for proper circular-arc graphs. (It should be remarked that W. L. Hsu has recently announced a linear-time algorithm for the representation of general interval graphs, which does not use PQ-trees; cf. [10].)

We have given an earlier algorithm for the representation of proper circular-arc graphs with time complexity $O(\Delta m)$ in [9]; cf. also [2, 8, 13]. (Here Δ denotes the

* Received by the editors November 25, 1992; accepted for publication (in revised form) September 7, 1994. Parts of this paper have appeared, in a preliminary form, in [6].

† Department of Computing Science, York University, North York, ON M3J 1P3, Canada.

‡ School of Computing Science, Simon Fraser University, Burnaby, BC V5A 1S6, Canada.

§ Department of Mathematics and Statistics, Simon Fraser University, Burnaby, BC V5A 1S6, Canada.

maximum degree of the input graph.) Thus the present algorithm is more efficient; nevertheless, for graphs of bounded degree both algorithms are linear, and we believe the algorithm in [9] (which is conceptually much simpler) remains interesting.

Our algorithm for the representation of proper circular-arc graphs has immediate applications in situations where existing algorithms for optimization problems on proper circular-arc graphs assume that a representation by circular-arcs is given. For instance, there are $O(n)$ algorithms for solving the maximum clique problem, the maximum independent set problem, and the q-colouring problem in proper circular-arc graphs, provided a representation by circular arcs is given; cf. [4, 11, 17]. In view of our $O(m + n)$ representation algorithm, we may now conclude that all these problems for proper circular-arc graphs are solvable in time $O(m+n)$ with no further restriction.

The correctness proof of our algorithms depends on the progress we made on another problem—the understanding of the structure of local tournaments. A *local tournament* (cf. [1, 8, 13]), is an oriented graph in which the inset as well as the outset of every vertex is a tournament. Local tournaments turn out to be a useful generalization of tournaments and many interesting algorithmic problems can be efficiently solved for this class of oriented graphs (cf. [1, 8, 13]). Local tournaments are relevant to our purposes because of the fact that (among connected graphs) proper circular-arc graphs are precisely the graphs orientable as local tournaments and proper interval graphs are precisely the graphs orientable as nonstrong local tournaments. These results are essentially due to Skrien [16] (cf. also [9, 12, 13]), and they allow us to deal with orientations of the input graph instead of having to deal with representations of it. We have developed new characterizations of local tournaments and of nonstrong local tournaments, and we shall use them below to prove the correctness of our algorithms.

To capture the situations where some edges of the input graph have already been oriented and others are still undirected, we employ the notion of a *mixed graph*: it is a graph H with some directed edges (arcs) and some undirected edges such that if H contains the directed edge xy, then it contains neither the directed edge yx nor the undirected edge xy. The *inset*, respectively the *outset*, of a vertex v in a mixed graph H is the set of all vertices u in H for which uv, respectively vu, is a directed edge of H. A directed path in a mixed graph contains only directed edges (all oriented in the same direction). Note that the class of mixed graphs without directed edges is precisely the class of undirected graphs and the class of mixed graphs without undirected edges is precisely the class of oriented graphs.

Let D be a mixed graph. If xy is a directed edge of D, then we say that x dominates y and write $x \rightarrow y$; otherwise, we say that x does not dominate y and write $x \not\rightarrow y$. We say that D is *strong* if, for any two vertices x and y, D contains a directed path from x to y (and a directed path from y to x); otherwise, it is called *nonstrong.*

Let G be an undirected graph. For any vertex v, let $N(v)$ be the neighbourhood of v, i.e., the set of vertices which are adjacent to v. The *closed neighbourhood* of v is the set $N[v] = N(v) \cup \{v\}$. We define an equivalence relation on $V(G)$ in which a and b are equivalent just if $N[a] = N[b]$. The classes of this equivalence are called the *blocks* of G. It follows from the definition that two blocks are either completely adjacent (every vertex of one is adjacent to every vertex of the other) or completely nonadjacent (no vertex of one is adjacent to any vertex of the other). We shall use the terms adjacent blocks and nonadjacent blocks to describe these respective situations. If the vertices a and b of an edge ab are equivalent, we call the edge ab *balanced*; otherwise, ab is an *unbalanced* edge. We say that G is *reduced* if there are no balanced edges, i.e., if distinct vertices have distinct closed neighbourhoods.

The *underlying graph* of a mixed graph D is the undirected graph $G(D)$ with the vertex set $V(D)$ in which xy is an edge of $G(D)$ only if it is a directed or undirected edge of D. We say that D is *connected* if $G(D)$ is connected. We say that D is *reduced* if $G(D)$ is reduced. We call an arc ab of D balanced if the edge ab is balanced in $G(D)$.

2. Local tournaments. First we discuss the structure of local tournaments. A detailed treatment appears in [12] (cf. also [13]); we summarize here only the main points relevant to our algorithm. We begin with nonstrong local tournaments.

A *straight enumeration* of an oriented graph D is a linear ordering $v_1, v_2, \dots, v_n$ of its vertices such that for each i there exist nonnegative integers k and l such that the vertex v_i has inset $\{v_{i-1}, v_{i-2}, \dots, v_{i-k}\}$ and outset $\{v_{i+1}, v_{i+2}, \dots, v_{i+l}\}$. Note that any arc v_iv_j has $i < j$ and the subgraph induced by $v_i, v_{i+1}, \dots, v_j$ is a transitive tournament. An oriented graph which admits a straight enumeration is called *straight.* An undirected graph is said to have a *straight orientation* if it admits an orientation which is a straight oriented graph.

PROPOSITION 2.1. *The following properties are equivalent for an oriented graph D:*

1. *D is straight*;

2. *there exists an inclusion-free family of intervals associated with the vertices of D such that u dominates v in D if and only if the interval associated with u contains the left endpoint of the interval associated with v (the interval of u intersects the interval of v "on the left").*

Proof. 1 *implies* 2: Given a straight enumeration $v_1, v_2, \dots v_n$ of D, we associate with v_i the interval from i to $i + d_i + 1 - \frac{1}{i}$, where d_i denotes the outdegree of v_i.

2 *implies* 1: Given an inclusion-free family of intervals, we order the vertices as $v_1, v_2, \dots, v_n$ in such a way that the left endpoints of the corresponding intervals form an increasing sequence. □

COROLLARY 2.2. *A graph is a proper interval graph if and only if it has a straight orientation.* □

We see from the above that we can construct the intervals of a representation of G in time $O(m + n)$ provided that we have a straight enumeration of an orientation D of G. Thus our algorithm for representing a proper interval graph needs only to find a straight enumeration of an orientation of G.

Note that a straight oriented graph is a nonstrong local tournament. (In fact, it is an acyclic local tournament.) Moreover, *all* nonstrong local tournaments arise in a simple fashion from straight oriented graphs. Let S be an oriented graph and let T_v be a family of disjoint tournaments indexed by the vertices v of S. The *substitution operation* on S with respect to T_v, $v \in V(S)$ results in an oriented graph D obtained from S by replacing each v by the corresponding T_v and with each vertex of T_v dominating each vertex of T_u in D just if $v \rightarrow u$ in S. Note that each edge of each T_v is balanced and, if S is reduced, there are no other balanced edges in D. The *full reversal* of an oriented graph D is the operation of reversing the directions of all oriented edges (arcs) of D.

The following theorem describes a method to generate all possible nonstrong local tournaments (other than tournaments) as well as all possible nonstrong local-tournament orientations of a fixed undirected graph.

THEOREM 2.3 ([12, 13]). *Let D be a connected oriented graph which is not a tournament. Then D is a nonstrong local tournament if and only if it is obtained from* *a reduced straight oriented graph S (with $|V(S)| > 1$) by the substitution operation with respect to some family of tournaments $T_v, v \in V(S)$.*

Moreover, if D is a nonstrong local tournament, then for every nonstrong local-tournament orientation D' of $G(D)$, either D' or the full reversal of D' is obtained from the same S by the substitution operation with respect to a family $T'_v, v \in V(S)$, where $|T'_v| = |T_v|$ for each $v \in V(S)$. □

COROLLARY 2.4. *Let G be a connected proper interval graph and D and D' two arbitrary nonstrong local tournament orientations of G. Then D' can be obtained from D by changing the directions of some balanced arcs and then possibly performing a full reversal.* □

In order to concisely describe all possible nonstrong local tournament orientations of a fixed connected proper interval graph, we introduce a modified substitution operation in which we replace each vertex v by a complete undirected graph T_v, again with each vertex of T_v dominating each vertex of T_u if and only if $v \rightarrow u$. A *straight mixed graph H* is a mixed graph obtained from a reduced straight oriented graph S by such a substitution operation with respect to a family T_v of complete graphs. Note that the blocks of the resulting mixed graph are precisely the vertex sets of the complete graphs T_v. (In other words, each undirected edge is balanced and each directed edge is unbalanced.) Let H be a straight mixed graph. *A straight enumeration* $V_1, V_2, \ldots, V_p$ of the blocks of H is an ordering of the blocks of H in the order of the straight enumeration of the corresponding vertices of S. In the following, we shall often define a straight mixed graph by describing a straight enumeration of its blocks. Given a sequence of blocks $V_1, V_2, \ldots, V_p$ of an undirected graph G, consider the mixed graph H obtained by leaving all edges xy of G with x and y from the same V_j undirected and orienting each edge xy with $x \in V_i$, $y \in V_j$, and $i < j$ from x to y. Then this is a straight mixed graph provided that for any i and j such that V_i is adjacent to V_j, the subgraph induced by the intermediate blocks $V_i \cup V_{i+1} \cup \cdots \cup V_j$ is complete.

If we orient the undirected edges so that each T_v becomes a transitive tournament, we obtain a straight oriented graph. Conversely, starting from a straight oriented graph, we obtain a straight mixed graph by replacing all balanced arcs by the corresponding undirected edges. Therefore we may extract from Corollaries 2.2 and 2.4 the following useful result.

COROLLARY 2.5. *Each connected proper interval graph is uniquely orientable as a straight mixed graph up to full reversal.* □

A similar discussion applies to general (possibly strong) local tournaments. A *round enumeration* of an oriented graph D is a circular ordering $v_1, v_2, \ldots, v_n$ of its vertices such that for each i there exist nonnegative integers k and l such that the vertex v_i has inset $\{v_{i-1}, v_{i-2}, \ldots, v_{i-k}\}$ and outset $\{v_{i+1}, v_{i+2}, \ldots, v_{i+l}\}$; here additions and subtractions are modulo n. Note that in this case there can be arcs v_iv_j with both $i < j$ and $i > j$; however, it is still true that if v_iv_j is an arc (regardless of whether $i < j$ or $j < i$), the subgraph induced by $v_i, v_{i+1}, \ldots, v_j$ is a transitive tournament. (The subscripts are again computed modulo n.) An oriented graph which admits a round enumeration is called *round*. An undirected graph is said to have a *round orientation* if it admits an orientation which is a round oriented graph. Clearly a round oriented graph is a local tournament. It is easy to see that an induced subgraph of a round oriented graph is a round oriented graph and a straight subgraph of a straight oriented graph is a straight oriented graph. Moreover, it is clear that a straight enumeration is a special case of a round enumeration and that, in fact, a round oriented graph is straight if and only if it is nonstrong.

PROPOSITION 2.6. *The following properties are equivalent for a connected ori-*

ented graph D:

1. *D is round;*

2. *there exists an inclusion-free family of circular arcs associated with the vertices of D such that $u \rightarrow v$ in D if and only if the circular arc associated with u contains the counterclockwise endpoint of the circular arc associated with v.* □

COROLLARY 2.7. *A connected graph is a proper circular-arc graph if and only if it has a round orientation.* □

The proof of Proposition 2.6 is analogous to that of Proposition 2.1, and it also yields an $O(m+n)$ method to find a representation of a given graph G by circular arcs, provided that a round enumeration of an orientation of G is given. In fact, we may modify the interval representation given in the proof of Proposition 2.1 by identifying two vertices x and y of the real line whenever $|x-y| = n+1$. This makes the real line into a circle and the intervals into circular arcs. Thus it will be sufficient to specify a linear-time algorithm to provide a round enumeration of an oriented graph D with $G(D) = G$ if one exists and to report that one does not exist otherwise.

Just as nonstrong local tournaments are related to straight oriented graphs in Theorem 2.3, general local tournaments are related to round oriented graphs. However, the relationship is more complicated. The next theorem describes a method to generate all possible local tournaments as well as all possible local-tournament orientations of a fixed undirected graph. A *special reversal* of a round oriented graph D is defined as follows: Consider the complement $\overline{G}$ of $G(D)$. If $\overline{G}$ has an odd cycle or if $\overline{G}$ is connected, then no special reversal is possible. Otherwise ($\overline{G}$ is disconnected and bipartite), we let A be either the set of all unbalanced arcs within a fixed component of $\overline{G}$ or the set of all unbalanced arcs between two fixed components of $\overline{G}$. The special reversal corresponding to A is the operation that reverses the directions of all arcs in the set A.

THEOREM 2.8 ([12, 13]). *Let D be a connected oriented graph. Then D is a local tournament if and only if it is obtained from some reduced round oriented graph R by the substitution operation with respect to some family $T_v, v \in V(R)$, possibly followed by special reversals.*

Moreover, if D is a local tournament, then every local-tournament orientation of $G(D)$ is obtained from the same R by the substitution operation with respect to a family $T'_v, v \in V(R)$, where $|T'_v| = |T_v|$ for each $v \in V(R)$, possibly followed by special reversals and/or a full reversal. □

COROLLARY 2.8. *Let G be a connected proper circular-arc graph and D and D' be two arbitrary local-tournament orientations of G. Then D' can be obtained from D by changing the directions of some balanced arcs and then possibly performing special reversals and/or a full reversal.* □

A *round mixed graph H* is a mixed graph obtained from a reduced round oriented graph R by such a substitution operation with respect to a family T_v ($v \in V(R)$) of complete graphs. Note that if we orient the undirected edges of a round mixed graph in an arbitrary way, we obtain a local tournament, and if we do it in such a way that each T_v is a transitive tournament, then we get a round oriented graph. Conversely, if we erase the directions of all balanced arcs in a round oriented graph, we obtain a round mixed graph. Thus we obtain the following result from Corollaries 2.7 and 2.9 (using the fact that no special reversals are possible if the complement of G is connected or nonbipartite).

COROLLARY 2.9. *Let G be a connected proper circular-arc graph. If the complement of G is connected or nonbipartite, then it is uniquely orientable as a round mixed graph up to full reversal.* □

3. Proper circular-arc graphs. We now give our linear-time algorithm for the representation of proper circular arc graphs. Recall that Tucker [18] gives an $O(n^2)$ algorithm; thus we only need to deal with the case when the number of edges is small relative to n^2. Let x be a vertex of minimum degree in G. Let A be the graph induced by $N[x]$ and let $B = G - A$. We may assume that B is not a complete graph. Indeed, if B is complete, we can find a representation of G by Tucker's algorithm; since x is a vertex of minimum degree, we conclude that the number of edges of G is $m \geq \frac{n^2-2n}{4}$ and so $n^2 = O(m+n)$, i.e., Tucker's algorithm is in fact linear in this case. We may also assume that G is not a proper interval graph—this can be tested in linear time by the algorithm presented in the next section. Note that a proper circular-arc graph which is not a proper interval graph must be connected.

Thus let G be a proper circular-arc graph which is not a proper interval graph. Note that this implies that G is connected and can be oriented as a strong round oriented graph $\vec{G}$. As mentioned above, we wish to find some round orientation of G. We assume from now on that B is not complete.

PROPOSITION 3.1. *Both A and B are connected proper interval graphs.*

Proof. Let $v_1, v_2, \ldots, v_n$ be a round enumeration of $\vec{G}$ and let $x = v_1$. Let f be the least integer such that v_f dominates x in $\vec{G}$ and let g be the largest integer such that x dominates v_g in $\vec{G}$. Then $f > g$ and both the subgraph of $\vec{G}$ induced by $\{v_f, v_{f+1}, \ldots, v_1\}$ and the subgraph of $\vec{G}$ induced by $\{v_1, v_2, \ldots, v_g\}$ are transitive tournaments; hence $A = \{v_f, v_{f+1}, \ldots, v_n, v_1, v_2, \ldots, v_g\}$ and $B = \{v_{g+1}, \ldots, v_{f-1}\}$. It follows that A and B are connected graphs. Let $\vec{A}$ be the subgraph of $\vec{G}$ induced by the vertices of A and let $\vec{B}$ be defined similarly. Then both $\vec{A}$ and $\vec{B}$ are round oriented graphs. We shall argue that they are, in fact, straight oriented graphs, and hence A and B are proper interval graphs. We shall show that, in fact, both A and B are acyclic oriented graphs. The subgraphs of A induced by $\{v_f, v_{f+1}, \ldots, v_1\}$ and by $\{v_1, v_2, \ldots, v_g\}$ are transitive tournaments oriented from v_f to v_1 and from v_1 to v_g. Thus for A it only remains to verify that there is no arc $v_i v_j$ with $2 \leq i \leq g$ and $f \leq j \leq n$. Any such arc would imply that $v_i, v_{i+1}, \ldots, v_j$ is a transitive tournament, contradicting the assumption that B is not complete. This proves that $\vec{A}$ is an acyclic round oriented graph and thus a straight oriented graph. For $\vec{B}$ the proof is similar, as any arc $v_i v_j$ with $g < j < i < f$ in $\vec{B}$ would imply that v_i dominates $v_1 = x$, contradicting the definition of B. □

According to the above proposition, we may construct a straight mixed orientation B' of B. For this, we may use the technique of the next section, which will also produce a straight enumeration $B_1, \ldots, B_q$ of the blocks of B'. Let L be the set of all vertices of A which are adjacent (in G) to a vertex of B_q and R the set of all vertices of A which are adjacent to a vertex of B_1. Let $C = G - L$ and $D = G - R$. Finally, let E be the subgraph of G induced by the vertices of $A \cup B_q$. Note that C and D both contain B as an induced subgraph and that E contains A as an induced subgraph.

PROPOSITION 3.2. *All three graphs C, D, and E are connected proper interval graphs.*

Proof. Consider again the round enumeration $v_1, v_2, \ldots, v_n$ of $\vec{G}$, where $x = v_1, A = \{v_f, v_{f+1}, \ldots, v_n, v_1, v_2, \ldots, v_g\}$, and $B = \{v_{g+1}, \ldots, v_{f-1}\}$. We claim that

$$C = \{v_{h+1}, v_{h+2}, \ldots, v_n, v_1, \ldots, v_{f-1}\}$$

for some $h, f \leq h \leq n$. Consider first the straight enumeration $v_{g+1}, \ldots, v_{f-1}$ of $\vec{B}$. Clearly, any block of $\vec{B}$ consists of consecutive vertices in this enumeration. Thus we obtain a straight mixed orientation of B by keeping the directions of all unbalanced edges of B according to their direction in $\vec{B}$. Since the B' is unique, up to a full reversal, we may assume, by considering a full reversal of $\vec{G}$ if necessary, that $B_q = \{v_s, v_{s+1}, \dots, v_{f-1}\}$ for some $s, g+1 < s < f$. Now we prove that $L = \{v_f, v_{f+1}, \dots, v_h\}$, where h is the largest integer such that v_{f-1} dominates v_h (thus $f \le h \le n$). Indeed, it follows from the definition of round enumeration that each $v_i, f \le i \le h$ is adjacent to a vertex (namely v_{f-1}) of B_q. On the other hand, if some v_j is dominated by a vertex in B_q, then it is also dominated by v_{f-1}; hence $f \le j \le h$ by the definition of h. Also, no v_j in A can dominate a vertex in B_q because then B would be a complete graph (recall that all vertices in B_q have the same closed neighbourhood in B). Having found L, we now know that $C = \{v_{h+1}, v_{h+2}, \dots, v_n, v_1, \dots, v_{f-1}\}$. In the order $v_{h+1}, v_{h+2}, \dots, v_n, v_1, \dots, v_{f-1}$ of vertices, there are no arcs from a later vertex to an earlier vertex because $v_{h+1}, v_{h+2}, \dots, v_n, v_1, \dots, v_g$ is part of the straight enumeration of $\vec{A}$, $v_{g+1}, v_{g+2}, \dots v_{f-1}$ is our straight enumeration of $\vec{B}$, and any arc $v_i v_j$ with $v_i \in B$ and $v_j \in A, j \notin \{f, f+1, \dots, h\}$ would imply that $v_i, v_{i+1}, \dots, v_j$ is a transitive tournament, contradicting the definition of h. Let $\vec{C}$ be the subgraph of $\vec{G}$ induced by the vertices of C. Then $v_{h+1}, v_{h+2}, \dots, v_n, v_1, \dots, v_{f-1}$ is a straight enumeration of $\vec{C}$. Hence C is a proper interval graph.

The argument for D is symmetric and it also yields the fact that $R = \{v_k, v_{k+1}, \dots, v_g\}$, where $1 < k \le g$. In particular, we note for future reference that the sets L and R are disjoint.

Let $\vec{E}$ be the subgraph of $\vec{G}$ induced by the vertices of E. Recall that $B_q = \{v_s, v_{s+1}, \dots, v_{f-1}\}$. We claim that $v_s, v_{s+1}, \dots, v_n, v_1, \dots, v_g$ is a straight enumeration of $\vec{E}$. This follows easily from the fact that $v_f, v_{f+1}, \dots, v_n, v_1, \dots, v_g$ is a straight enumeration of $\vec{A}$, $v_s, v_{s+1}, \dots, v_{f-1}$ a part of our straight enumeration of $\vec{B}$, and there is no arc $v_i v_j$ with $v_i \in A$ and $v_j \in B_q$ because B is not complete. The connectivity of C, D, E is clear from the above arguments. □

Now we may compute, again using techniques of the next section, straight mixed orientations C', D', E' of C, D, E, respectively. Recall that these orientations are unique up to full reversal. Our algorithm consists of combining the straight mixed graphs C', D', E' to form a round mixed orientation of G. To do this, we may have to replace some of C', D', E' by their full reversals. It must be possible to do this *consistently*, i.e., in such a way that all edges of G which are oriented in more than one of the mixed graphs C', D', E' are oriented there in the same way. To see this, consider the straight mixed graphs C'', D'', E'' obtained from $\vec{C}, \vec{D}, \vec{E}$ by replacing all balanced arcs by the corresponding undirected edges. They clearly have the property that all edges of G are oriented consistently (because they all arise from $\vec{G}$); however, according to Proposition 3.2 and Corollary 2.5, each of C', D', E' is equal to the corresponding C'', D'', E'' or its full reversal. To explicitly find the right orientations C', D', E', we may proceed as follows. Arbitrarily fix C' to be either of the two possible straight mixed orientations of C. This determines D', since C and D both contain B, which is not complete and hence contains an unbalanced arc. (In other words, fixing an orientation of C determined the direction of at least one edge of D and hence determined D'.) In turn, this determines the orientation of E since the edge $v_{f-1}v_f$ of G is an unbalanced edge of both D and E (recall that x is not adjacent to v_{f-1} but is adjacent to v_f). It is clear that these operations can be performed in time $O(m+n)$ once any straight mixed orientations of C, D, E are known. Thus assume that C', D', E' are oriented consistently and let H be the mixed orientation of G in which an edge is oriented only if it is oriented in any of the graphs C', D', E' and is oriented according to the orientation it has there; clearly, this can also be done in time $O(m+n)$.

PROPOSITION 3.3. *The mixed graph H is a round mixed graph.*

Proof. We first prove that each unbalanced edge of G is unbalanced in one of the mixed straight graphs C', D', E' (and hence it is oriented in that graph). Consider an unbalanced edge uv of G. If both u and v are vertices of B and if uv is balanced in B, then there exists a vertex $z \in A$ such that, say, z is adjacent to u but not to v. Since $L \cap R = \emptyset$ (see the last sentence of the proof of Proposition 3.2), $z \in C$ or $z \in D$. Thus u, v, z are vertices of C or D and hence the edge uv is an unbalanced edge in C or D. If uv is unbalanced in B, then it is also unbalanced in C (and D) since B is an induced subgraph of C. If both u and v are vertices of A and if uv is balanced in A, then there exists a vertex $z \in B$ such that, say, z is adjacent to u but not to v. Suppose first that z dominates u. If any vertex of B dominates u, then a vertex z' of B_q dominates u (consider the round enumeration $\dots, z, \dots, v_{f-2}, v_{f-1}, \dots, u, \dots$ of $\vec{G}$); thus $u \in L$. If z' is adjacent to v, then u, v, z are vertices of D and uv is an unbalanced edge of D; if z' is not adjacent to v, then u, v, z' are vertices of E and uv is an unbalanced edge of E. If uv is unbalanced in A, then it is also unbalanced in E. If (say) u is a vertex of A and v a vertex of B, then x is adjacent to u but not v and $u \notin L$ or $u \notin R$; thus u, v, x are vertices of C or D and uv is an unbalanced edge of C or D. From the way A', B', C', D', E' were chosen, we see that they all agree with $\vec{G}$ or the full reversal of $\vec{G}$. Therefore, H is obtained from $\vec{G}$ (or its reversal) by ignoring the directions on the balanced arcs. Therefore, H is a round mixed graph. □

We now summarize the algorithm.

ALGORITHM 3.4. *Let G be a graph with n vertices and m edges.*

[Step 1.] *Test if G is a proper interval graph by Algorithm* 4.4. *If it is, then represent it by linear intervals viewed as a special case of circular arcs.*

[Step 2.] *Choose a vertex x of minimum degree in G. Let A be the graph induced by $N[x]$ and let $B = G - A$. If B is a clique, then find a representation of G by Tucker's algorithm. Otherwise, proceed as follows.*

[Step 3.] *If B is not a proper interval graph, then report that G is not a proper circular-arc graph. Otherwise, orient B as a straight mixed graph B' by Algorithm* 4.4, *which also produces a straight enumeration $B_1, B_2, \dots, B_q$ of the blocks of B'.*

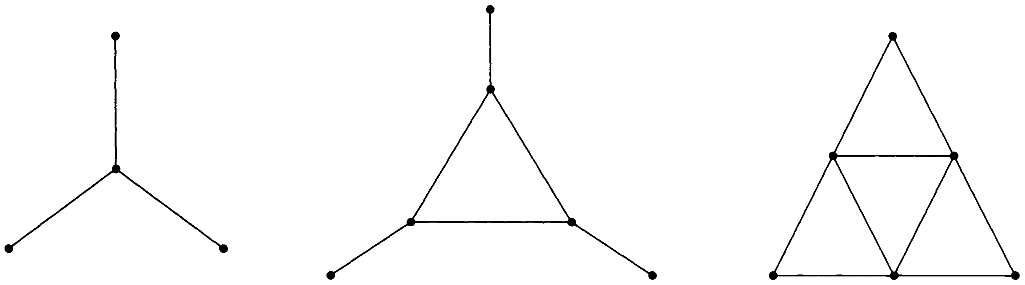
[Step 4.] *Let L be the set of vertices of A adjacent to a vertex of B_q and let R be the set of vertices of A adjacent to a vertex of B_1. Let $C = G - L$ and $D = G - R$, and let E be the subgraph of G induced by $A \cup B_q$. If any of C, D, E is not a proper interval graph, then report that G is not a proper circular-arc graph. Otherwise, orient C, D, E as straight mixed graphs C', D', E' by Algorithm* 4.4.

[Step 5.] *If it is not possible to perform full reversals on C', D', E' so that all edges are oriented consistently, then report that G is not a proper circular-arc graph. Otherwise find consistent orientations C', D', E' and construct H from G by orienting any edge uv so that $u \rightarrow v$ provided that $u \rightarrow v$ in any of the mixed graphs C', D', E'.*

[Step 6.] *Orient all remaining undirected edges of H so that each block of H becomes a transitive tournament. Let D be the resulting oriented graph.*

[Step 7.] *Transform D into a circular-arc representation of G by the method of Proposition* 2.6.

The correctness of the algorithm is implicit in our propositions. When we report that G is not a proper circular-arc graph in Step 3, then Proposition 3.1 implies that this is true. When we report this in Step 4, then Proposition 3.2 implies that this is true. When we report the same conclusion in Step 5, then this is true by the remarks following the proof of Proposition 3.2. The mixed graph H constructed in Step 5



the claw the net the tent

FIG. 1. *The claw, the net, and the tent.*

is round by Proposition 3.3. The oriented graph D constructed in Step 6 is round according to the remark following Corollary 2.8.

Steps 1, 3, and 4 are done in linear-time by Algorithm 4.4. (In Step 4, we also need to identify L and R, but this can clearly be done in time $O(m+n)$.) Step 2 also takes only linear-time according to the comments at the beginning of this section. We have already explained how the test for consistency can be done in time $O(m+n)$. The remainder of Step 5 and Step 6 are clearly linear. Step 7 can also be carried out in linear-time, as explained after Proposition 2.6.

4. Proper interval graphs.

In this section, we give an $O(m+n)$-time algorithm to represent proper interval graphs. As mentioned above, such algorithms already exist. However, our algorithm is simpler than the existing algorithms, avoids difficult data structures, and directly produces the straight mixed graph orientation of the input graph that is required by Algorithm 3.4. Furthermore, the proof of correctness of our algorithm implies as a byproduct a new proof of a theorem of Wegner [19]; cf. Corollary 4.3.

Assume that G is a connected graph, as otherwise we can work separately on each component of G. Our algorithm will insert vertices of G one at a time into an already formed straight mixed graph to form a new straight mixed graph. If G is a proper interval graph, then this process continues successfully until a straight mixed graph orientation of G is obtained. This is what is needed in Algorithm 3.4. If a concrete representation by intervals is desired, then we can again orient the remaining undirected edges so that each block is a transitive tournament and then represent the resulting straight oriented graph as explained in the proof of Proposition 2.1. (All this can still be done in time $O(m+n)$.) It is easy to see that any chordless cycle of length greater than three and any of the three graphs (the claw, the net, or the tent) in Fig. 1 is not orientable as nonstrong local tournament and hence is not a proper interval graph. (This is also well known [7].) Therefore, no graph which contains a chordless cycle of length greater than three, a copy of the claw, the net, or the tent, as an induced subgraph can be a proper interval graph.

We emphasize that by Corollary 2.5 the straight mixed graph orientation of a connected proper interval graph is unique up to a full reversal. Therefore, the corresponding straight enumeration of blocks is also unique up to reversing the order of the blocks. This is crucial in what follows, even though it is not always explicitly mentioned. In particular, if we have two connected straight mixed graphs H and H' with straight enumerations $V_1, V_2, \dots, V_p$ and $V_1', V_2', \dots, V_{p'}'$ and if H is an induced subgraph of H', then we may assume (by performing a full reversal on H' if necessary) that each edge oriented both in H and H' is oriented in the same directions in both. Moreover, it is clear that two vertices equivalent in H' are also equivalent in H. Therefore, any edge unbalanced in H remains unbalanced in H'; this also means that any edge oriented in H remains oriented in H'. Suppose that H' has just one more vertex than H, say vertex v. Then each V_i breaks into two blocks $V_i \cap N(v)$ and $V_i - N(v)$ of H' (if v has both some neighbours and some nonneighbours in V_i), combines with v to form a block $V_i \cup \{v\}$ of H' (if v has exactly the same closed neighbourhood as V_i in H'), or remains a block of H' (otherwise).

Now we suppose that G is a connected proper interval graph and H is a straight mixed orientation of a connected subgraph of G. Let $V_1, V_2, \ldots, V_p$ be a fixed straight enumeration of the blocks of H. Let v be a vertex of G which is not in H but is adjacent to a vertex of H. We wish to find a straight mixed orientation H' of the subgraph of G induced by v together with the vertices of H. According to the above observations on the form of the blocks of H', it remains only to describe which edges of G not oriented in H are oriented in H'. We can do this by describing the straight enumeration of the blocks of H', because the direction of all unbalanced edges is determined by this order, to go from the earlier block towards the later block.

Before we can describe the straight enumeration of the blocks of H', we need to analyze the possible connections of v in H. In fact, these are rather restricted as far as the blocks of H are concerned.

PROPOSITION 4.1. *Let* $1 \leq a < b < c \leq p$.

1. *If* v *is adjacent to a vertex in* V_a *and to a vertex in* V_c*, then* v *is completely adjacent to* V_b *(i.e.,* v *is adjacent to every vertex of* V_b*).*

2. *If* v *is adjacent to a vertex in* V_b *and nonadjacent to a vertex in* V_a *and a vertex in* V_c*, then* V_a *is nonadjacent to* V_c.

3. *If* $V_a \rightarrow V_b \rightarrow V_c$ *and if* v *is adjacent to a vertex in* V_b*, then* v *is completely adjacent to* V_a *or to* V_c.

Proof. Assume that v is adjacent to $x \in V_a$ and $z \in V_c$ but not to $y \in V_b$. We may assume without loss of generality that a, b, and c are chosen so that $c - a$ is minimal. Then $c - a \geq 2$ and v is not adjacent to at least one vertex in each V_d with $a < d < c$. Moreover, V_a and V_c are adjacent blocks, for otherwise any shortest path from V_a to V_c in $V_a \cup V_{a+1} \cup \cdots \cup V_c$ together with v would induce chordless cycle of length greater than three, a contradiction. (The subgraph induced by $V_a \cup V_{a+1} \cup \cdots \cup V_b$ is connected because the connectivity of H implies that each V_i is adjacent to V_{i+1}.) Since $V_1, V_2, \cdots, V_p$ is a straight enumeration of H, the vertices in $V_a \cup V_{a+1} \cup \cdots \cup V_b$ induce a complete subgraph of G. Consider the three blocks V_a, V_b, V_c; any two distinct blocks have distinct neighbourhoods in H. Thus there exists a block V_i which is adjacent to exactly one of the blocks V_a, V_b, and there exists a block V_j which is adjacent to exactly one of the blocks V_b, V_c. If V_i is adjacent to V_a but not to V_b and V_j is adjacent to V_c but not to V_b, then clearly $i < a < c < j$, and v must be completely adjacent to both V_i and V_j, as otherwise G would contain a copy of the claw centered at x or z. Now the vertices x, y, z, v together with any vertex of V_i and any of vertex of V_j induce a copy of the tent, contradicting the fact that G is a proper interval graph. Thus we may assume without loss of generality that V_i is adjacent to V_b but not to V_a. Then clearly $i > c$ and v must be completely nonadjacent to V_i, as otherwise G would contain a four-cycle induced by v, x, z and a vertex of V_i. If, additionally, V_j is adjacent to V_b but not to V_c, then we would similarly have $j < a$ and v completely adjacent to V_j. In this case G would contain a copy of the tent induced by x, y, z, v, any vertex of V_i, and any vertex of V_j. If, on the other hand, V_j is adjacent to V_c but not to V_b, then clearly $c < i < j$ and, moreover, v must be completely adjacent to V_j, as otherwise G would contain a copy of the claw centered at z. Now we conclude that G contains a chordless five-cycle induced by v, x, y, any vertex of V_i, and any vertex of V_j.

For the second statement, suppose that v is adjacent to $y \in V_b$ and is nonadjacent to $x \in V_a$ and $z \in V_c$. Assume that V_a is adjacent to V_c. The blocks V_a and V_b have distinct closed neighbourhoods in H. If there is a block V_d which is adjacent to V_b but not adjacent to V_a, then $d > c$ because V_a is adjacent to V_c. Note that v is not adjacent to any vertex in V_d according to the first statement. Hence for any $w \in V_d$, $\{x, y, w, v\}$ induces a claw in G, contradicting the fact that G is a proper interval graph. Thus there must be a block V_e which is adjacent to V_a but nonadjacent to V_b. Similarly, there is a block V_f which is adjacent to V_c but nonadjacent to V_b. Note that $e < a$ and $f > c$. Hence v is adjacent to no vertex in V_e or V_f. Choose any $u \in V_e$ and any $w \in V_f$ and note that the subgraph induced by $\{x, y, z, u, w, v\}$ is a copy of the net, again contradicting the fact that G is a proper interval graph.

For the last statement, suppose that there are three vertices $x \in V_a$, $y \in V_b$, and $z \in V_c$ such that v is adjacent to y but not to x or z. By the second statement, x is not adjacent to z. Then G contains a claw induced by $\{x, y, z, v\}$. □

Remark. We note that the proof of Proposition 4.1 applies even if we replace the assumption that G is a connected proper interval graph by the assumption that G is a connected chordal graph which contains no induced claw, net, or tent. (A graph is called *chordal* if it contains no chordless cycle of length greater than three [7].) □

Under the above assumptions that G is a connected proper interval graph (or indeed that G is a connected chordal graph which contains no induced claw, net, or tent), H a straight mixed orientation of a connected subgraph of G, and v a vertex of G but not of H, we also have the following.

PROPOSITION 4.2. *The subgraph of G induced by v and the vertices of H is a proper interval graph.*

Proof. According to part 1 of Proposition 4.1, the blocks of H that are completely adjacent to v form a (possibly empty) sequence of consecutive blocks, say, $V_l, V_{l+1}, \ldots, V_r$, for some l and r. We first assume that $1 < l < r < p$. Let $a = l - 1$ and $c = r + 1$; thus $a \geq 1$ and $c \leq p$. Note that (again by part 1 of Proposition 4.1), v is nonadjacent to all V_j with $j < a$ and $j > c$. It further follows from part 2 of Proposition 4.1 that V_a and V_c are nonadjacent. Let b be the largest integer greater than a such that V_b is adjacent to V_a, and let d be the smallest integer less than c such that V_d is adjacent to V_c. Then both $a < b < c$ and $a < d < c$. We claim that $b < d$. Suppose $b \geq d$ and let x and y be two vertices in V_a and V_c, respectively, which are nonadjacent to v; such vertices exist by the definition of l and r. Let z be any vertex of V_d. Then H has a claw induced by $\{x, y, z, v\}$.

We are now ready to describe the straight mixed graph H' in the case when $1 < l < r < p$. If the set $N(v)$ intersects V_a, then (cf. above) V_a breaks into $V_a - N(v)$ and $V_a \cap N(v)$ and we modify the straight enumeration of the blocks of H by replacing V_a with $V_a - N(v), V_a \cap N(v)$, in this order. Similarly, if the set $N(v)$ intersects V_c, then we modify the straight enumeration by replacing V_a with $V_c \cap N(v), V_c - N(v)$, in this order. Note that these operations imply that some of the previously undirected edges have become oriented, e.g., the edges oriented from $V_c - N(v)$ to $V_c \cap N(v)$. It remains to describe where the vertex v goes in the straight enumeration of H'. Recall that it can form its own block or it can be added to an existing block V_j.

If $N(v)$ intersects V_a, then we put $\{v\}$ as a separate block just after V_b. If $N(v)$ intersects V_c, then we put $\{v\}$ as a separate block just before V_d. We note that if $N(v)$ intersects both V_a and V_c, then these two instructions coincide because in this case we necessarily have $d = b + 1$. Suppose $d > b + 1$, and let x and y be vertices of $N(v) \cap V_a$ and $N(v) \cap V_c$, respectively. Let z be any vertex of V_{b+1}. Then $\{x, y, z, v\}$ forms a claw of G, a contradiction. (The vertices x, y are nonadjacent because z and y are nonadjacent and V_a precedes V_{b+1} in the straight enumeration of H_{i+1}.) It is not difficult to verify that what we have defined is indeed a straight enumeration of H'. For instance, suppose that both $N(v) \cap V_a$ and $N(v) \cap V_c$ are nonempty and consider two adjacent blocks from the straight enumeration

$$V_1, \ldots, V_{a-1}, V_a - N(v), V_a \cap N(v), \ldots, V_b, \{v\}, V_d, \ldots, V_c \cap N(v), V_c - N(v), V_{c+1}, \ldots, V_p$$

described above. If two blocks V_i and V_j with $i < j < a$ are adjacent, then the intermediate blocks are $V_i, V_{i+1}, \ldots, V_j$ and they induce a complete graph since this was the case in H; a similar argument applies if $i > j > c$. If $a < i < c$, then the intermediate blocks may additionally contain also the block $\{v\}$, which was not included in H. However, $\{v\}$ is adjacent to all blocks $V_i, V_{i+1}, \ldots V_j$ and the graph induced by the intermediate blocks is still complete. If the block $V_a - N(v)$ or the block $V_a \cap N(v)$ is adjacent to V_j, then $j \leq b$ by the definition of b and again the graph induced by the intermediate blocks is complete. If the block v is adjacent to a block V_j with, say, $j \leq b$, then it is adjacent to all blocks V_i with $j \leq i \leq b$ and once more the graph induced by the intermediate blocks is complete. A similar argument applies in the case of the adjacent blocks $\{v\}$ and $V_a \cap N(v)$. In what follows, we shall omit these simple verifications. They are all similar to the above.

If $N(v)$ intersects neither V_a nor V_c, then $N(v) = V_l \cup \cdots \cup V_r$. There may be a block V_j with $b < j < d$ adjacent to both V_l and V_r. Since by the definition of b and d we know that V_j is not adjacent to $V_{l-1} = V_a$ or $V_{r+1} = V_c$, the closed neighbourhood in H_i of any vertex of V_j is $V_l \cup \cdots \cup V_r$. Thus there can be at most one such block V_j, as distinct blocks have distinct closed neighbourhoods. Moreover, the closed neighbourhood of V_j in H' is the same as that of v. Thus we obtain a straight enumeration of H' by replacing V_j with $V_j \cup \{v\}$. On the other hand, each block V_j with $b < j < d$ is adjacent to at least one of V_l, V_r because otherwise we would have a claw formed by v and any vertices from V_l, V_j, V_r. Let u be the least integer greater than b such that V_u is adjacent to V_r and w be the largest integer smaller that d such that V_w is adjacent to V_l. Note that $b < u \leq d$ and $b \leq w < d$. We may assume that $u > w$ because otherwise there would be a block V_j with $b < j < d$ adjacent to both V_l and V_r. This means that $u = w + 1$ because each V_j with $b < j < d$ is adjacent to at least one of V_l, V_r. To obtain the desired straight enumeration of H' from $V_1, \ldots, V_p$, we insert a new block $\{v\}$ just before V_u.

The case $1 = l < r < p$ is very similar to the above arguments. We let $b = l = 1$ and $c = r + 1$ and define d as above to be the smallest integer less than c such that V_d is adjacent to V_c. Then the above discussion applies almost verbatim, except that we may have $d = b = 1$. In this case we insert the block $\{v\}$ before V_1.

Because of symmetry, it only remains to consider the following cases:

When $1 = l = r = p$, then H has only one block, V_1, and v is completely adjacent to it. Then $V_1 \cup \{v\}$ is the unique block of H'.

When $1 = l = r < p$, then $\{v\}, V_1, V_2 \cap N(v), V_2 - N(v), \ldots, V_p$ is a straight enumeration of the blocks of H'.

When $1 = l < r = p$, then v is completely adjacent to all blocks of H. We define $b = 1$ and $d = p$ and proceed as above, letting w be the largest integer greater than 1 such that V_w is adjacent to V_1 and u be the least integer smaller than p such that V_u is adjacent to V_p. Thus if $u = w$, we have the enumeration $V_1, \ldots, V_u \cup \{v\}, \ldots, V_p$ and if $u > w$, i.e., $u = w+1$ (cf. above), we have the enumeration $V_1, \ldots, V_w, \{v\}, V_u, \ldots, V_p$.

(Clearly, $u < w$ is not possible.)

The case $1 < l = r < p$ is impossible by part 3 of Proposition 4.1.

Finally, there is the trivial case when $l > r$, i.e., when v is not completely adjacent to any block of H. In the ordering the vertices of G as $v_1, \ldots, v_n$, we made sure that $v = v_{i+1}$ is adjacent to some vertex of H, say in block V_u. By part 3 of Proposition 4.1, $u = 1$ or $u = p$. Without loss of generality, we may assume that $u = 1$. If $p = 1$, i.e., if H has only one block V_1, then $\{v\}, V_1 \cap N(v), V_1 - N(v)$ is a straight enumeration of the blocks of H'. The case $p = 2$ is impossible since distinct blocks of H have distinct closed neighbourhoods, and when $p \geq 3$ then part 3 of Proposition 4.1 implies that v is nonadjacent to all V_j with $j > 1$. Then $\{v\}, V_1 \cap N(v), V_1 - N(v), V_2, \ldots, V_p$ is a straight enumeration of the blocks of H'. Now we have described the straight mixed graph H' in all cases. □

It is interesting to note that the proposition implies the following well-known result [7] (see also [3] for a refinement).

COROLLARY 4.3 ([19]). *Let G be an undirected graph. Then G is a proper interval graph if and only if it is chordal and does not contain the claw, the net, or the tent as an induced subgraph.*

Proof. We have already observed that the condition is necessary. Therefore, suppose that there is a chordal graph which does not contain an induced claw, net, or tent and which is not a proper interval graph, and let G be such a graph which has the smallest possible number of vertices. Then G must be connected; let v be a vertex of G such that $G - v$ is also connected. By the minimality of G, we know that $G - v$ can be oriented as a straight mixed graph H. Since Proposition 4.2 is valid for the graph G (cf. the sentence just before Proposition 4.2), we see that G itself is orientable as a straight mixed graph, contradicting the fact that G is not a proper interval graph. □

The proof of the above proposition is, in fact, an algorithm for inserting v into H. We summarize our algorithm for the representation of proper interval graphs as follows.

ALGORITHM 4.4. *Let G be a connected graph.*

[Step 1.] *Order the vertices of G as $v_1, v_2, \ldots, v_n$ in such a way that the subgraph induced by $\{v_1, v_2, \ldots, v_i\}$ is connected, for each $i = 1, 2, \ldots, n$. Let $v = v_1$ and $H = \{v\}$.*

[Step 2.] *Perform the following operation as long as possible: If the vertices of H are $\{v_1, v_2, \ldots, v_i\}$, then let $v = v_{i+1}$ and insert v into H as described above.*

[Step 3.] *If H does not contain all vertices, then report that G is not a proper interval graph.*

[Step 4.] *If desired, transform H into an interval representation of G by first orienting each block to be a transitive tournament and then using the method of Proposition* 2.1.

The correctness of the algorithm is assured by Proposition 4.2. Indeed, if the algorithm halts before H contains all the vertices, then some v could not be inserted and hence G is not a proper interval graph.

Our Algorithm 4.4 can be implemented in time $O(m + n)$. Since the first version of this manuscript, Corneil, Kim, Natarajan, and Olariu have published an even simpler linear-time recognition algorithm for proper interval graphs ("Simple linear time recognition of unit interval graphs," *Inform. Process. Lett.*, 55 (1995), pp. 99–104). It is possible to use their algorithm instead of ours for the purposes of Algorithm 3.4. One only has to notice that their algorithm can produce a straight mixed graph orientation of the input graph because our blocks B_i consist precisely of vertices v with equal value of their function $NextD(v)-PrevD(v)$. (Our Algorithm 4.4 remains interesting because it is incremental, and its correctness proof implies Wegner's theorem.)

Finally, we remark that we believe there is an incremental linear-time algorithm to directly compute a round mixed orientation of a given undirected graph along the lines of Algorithm 4.4. We hope to return to this idea in the future.

Acknowledgment. The authors wish to thank R. Shamir for his interest and for helpful comments.

REFERENCES

[1] J. Bang-Jensen, *Locally semicomplete digraphs: A generalization of tournaments*, J. Graph Theory, 14 (1990), pp. 371–390.

[2] J. Bang-Jensen, P. Hell, and J. Huang, *Local tournaments and proper circular arc graphs*, Technical report CSS/LCCR TR90-11, Simon Fraser University, Burnaby, BC, Canada, 1990.

[3] J. Bang-Jensen and P. Hell, *A note on chordal proper circular arc graphs*, Discrete Math., 128 (1994), pp. 395–398.

[4] B. Bhattacharya, P. Hell, and J. Huang, *A linear algorithm for maximum cliques in proper circular arc graphs*, SIAM J. Discrete Math., 9 (1996), to appear.

[5] K. S. Booth and G. S. Lueker, *Testing for the consecutive ones property using PQ-tree algorithms*, J. Comput. System Sci. 13 (1976), pp. 335–379.

[6] X. Deng, P. Hell, and J. Huang, *Recognition and representation of proper circular arc graphs*, in Integer Programming and Combinatorial Optimization, Proc. 2nd Integer Programming and Combinatorial Optimization Conference, E. Balas, G. Cornuejols, and R. Kannan, eds., Carnegie–Mellon University, Pittsburgh, PA, 1992, pp. 114–121.

[7] M. C. Golumbic, Algorithmic Graph Theory and Perfect Graphs, Academic Press, New York, 1980.

[8] P. Hell, J. Bang-Jensen, and J. Huang, *Local tournaments and proper circular arc graphs*, in Algorithms, Springer-Verlag Lecture Notes in Computer Science, T. Asano, T. Ibaraki, H. Imai, and T. Nishizeki, eds., Springer-Verlag, Berlin, New York, 1990, pp. 101–109.

[9] P. Hell and J. Huang, *Lexicographic orientation and representation algorithms for comparability graphs, proper circular arc graphs, and proper interval graphs*, J. Graph Theory, 20 (1995), pp. 361–374.

[10] W. L. Hsu, *A simple test of interval graphs*, in Proc. SIAM Conference on Discrete Mathematics, Vancouver, 1992.

[11] W. L. Hsu and K. H. Tsai, *Linear time algorithms on circular arc graphs*, Inform. Proc. Lett., 40 (1991), pp. 123–129.

[12] J. Huang, *On the structure of local tournaments*, J. Combin. Theory Ser. B, 63 (1995), pp. 200–221.

[13] ———, *Tournament-like oriented graphs*, Ph.D. thesis, Simon Fraser University, Burnaby, BC, Canada, 1992.

[14] N. Korte and R. H. Möhring, *An incremental linear-time algorithm for recognizing interval graphs*, SIAM J. Comput., 18 (1989), pp. 68–81.

[15] J. B. Orlin, M. A. Bonuccelli, and D. P. Bovet, *An $O(n^2)$ algorithm for coloring proper circular arc graphs*, SIAM J. Algebraic Discrete Meth., 2 (1981), pp. 88–93.

[16] D. J. Skrien, *A relationship between triangulated graphs, comparability graphs, proper interval graphs, proper circular arc graphs and nested interval graphs*, J. Graph Theory, 6 (1982), pp. 309–316.

[17] A. Teng and A. Tucker, *An $O(qn)$ algorithm to q-color a proper family of circular arcs*, Discrete Math., 55 (1985), pp. 233–243.

[18] A. Tucker, *Matrix characterization of circular arc graphs*, Pacific J. Math, 39 (1971), pp. 535–545.

[19] G. Wegner, *Eigenschaften der nerven homologische-einfactor familien in R^n*, Ph.D. thesis, Universität Gottingen, Gottingen, Germany, 1967.

AN $O(N+M)$-TIME ALGORITHM FOR FINDING A MINIMUM-WEIGHT DOMINATING SET IN A PERMUTATION GRAPH*

C. RHEE[†], Y. D. LIANG[‡], S. K. DHALL[§], AND S. LAKSHMIVARAHAN[§]

Abstract. Farber and Keil [*Algorithmica*, 4 (1989), pp. 221–236] presented an $O(n^3)$-time algorithm for finding a minimum-weight dominating set in permutation graphs. This result was improved to $O(n^2 \log^2 n)$ by Tsai and Hsu [*SIGAL '90 Algorithms, Lecture Notes in Computer Science*, Springer-Verlag, New York, 1990, pp. 109–117] and to $O(n(n+m))$ by the authors of this paper [*Inform. Process. Lett.*, 37 (1991), pp. 219–224], respectively. In this paper, we introduce a new faster algorithm that takes only $O(n+m)$ time to solve the same problem, where m is the number of edges in a graph of n vertices.

Key words. algorithm, dominating set, permutation graph

AMS subject classifications. 68Q20, 68Q25, 68R10

1. Introduction.

Let $\pi = (\pi[1], \pi[2], \ldots, \pi[n])$ be a permutation on the set $V_n = \{1, 2, \ldots, n\}$. For example, if $\pi = (2, 1, 5, 4, 6, 3)$, then $\pi[3] = 5, \pi[5] = 6, \pi^{-1}[5] = 3$, and $\pi^{-1}[6] = 5$, where $\pi^{-1}[i]$ denotes the *position* of the number i in the sequence $(\pi[1], \pi[2], \ldots, \pi[n])$. Let $G(\pi) = (V, E)$ be an undirected graph such that $V = V_n$ and $(i, j) \in E$ if and only if $(i-j)(\pi^{-1}[i] - \pi^{-1}[j]) < 0$. Then an undirected graph G is called a *permutation graph* if there is a permutation π such that G is isomorphic to $G(\pi)$[5].

For an undirected graph $G = (V, E)$, a subset D of V is called a *dominating set* for G if for every $u \in V - D$ there exists $v \in D$ such that $(u, v) \in E$, where $V - D$ denotes the set-theoretic difference of V and D. A graph G is called a *weighted graph* if there is a weight function $w : V \longrightarrow R$ (the set of reals), where $w[i]$ is called the weight of node i. Manacher and Mankus [7] showed that any algorithm for finding a minimum-weight dominating set for nonnegative weights can be extended to incorporate negative-weight vertices without loss of efficiency. Hence, for simplicity, we assume that $w[i]$ is nonnegative. For any subset $D \subseteq V$, let $W(D) = \sum_{i \in D} w[i]$ denote the *weight of D*. A subset $D \subseteq V$ is called a *minimum-weight dominating set* for G if $W(D)$ is a minimum over all dominating sets for G.

Permutation graphs are known to have a variety of practical applications [5], and for this reason, many algorithms have been developed in the literature. Pnueli, Lempel, and Even [8] described an $O(n^3)$ algorithm for testing if a given undirected graph G is a permutation graph. Spinrad [10] improved the above result by deriving an $O(n^2)$ algorithm for orienting comparability graphs. A proper subset of the permutation graphs called cographs has been studied by Corneil, Perl, and Stewart [3]. Bipartite permutation graphs were analyzed by Spinrad, Brandstädt, and Stewart [11]. Farber and Keil [4] developed $O(n^3)$ algorithms for finding a minimum-weight dominating set and a minimum-weight independent dominating set in permu-

* Received by the editors February 23, 1994; accepted for publication (in revised form) September 7, 1994. This research was supported in part by CAPS (Center for the Analysis and Prediction of Storms), an NSF Science and Technology Center at the University of Oklahoma.

[†] Department of Mathematics, Statistics, and Computer Science, Eastern Kentucky University, Richmond, KY 40475.

[‡] Department of Computer Science, Indiana Purdue University at Fort Wayne, Fort Wayne, IN 46805.

[§] School of Computer Science, University of Oklahoma, Norman, OK 73019.

tation graphs. In addition, they also described an $O(n^2)$ algorithm for the minimum cardinality dominating set problem. Atallah, Manacher, and Urrutia [2] presented an $O(n \log^2 n)$ algorithm for finding a minimum independent dominating set in a permutation graph. Later Atallah and Kosaraju [1] presented an $O(n \log n)$ time algorithm for the *maxdominance problem* to which the minimum-weight independent dominating set problem for permutation graphs can be reduced in linear time. Rhee, Dhall, and Lakshmivarahan [9] gave the first NC algorithm for minimum-weight domination in permutation graphs. Recently Tsai and Hsu [12] and the authors of this paper [6] improved one of Farber and Keil's results for the minimum-weight dominating set problem to $O(n^2 \log^2 n)$ time and $O(n(n+m))$ time, respectively. In this paper we propose an $O(n+m)$ time algorithm for finding a minimum-weight dominating set in a permutation graph G, where n and m indicate the number of vertices and edges in G, respectively.

In §2 we provide some preliminary definitions and state some theorems from [6] that lay out an essential dynamic computation structure of our algorithm. In §3 we present the algorithm and some data structures that support an efficient implementation of the algorithm. In §4 we derive time bounds of various subroutines used in the algorithm. Section 5 is devoted to describing an efficient scheme for computing a major component of the algorithm. A summary and time-complexity analysis of the algorithm is given in §6. Finally, some concluding remarks are made in §7.

2. Preliminaries. For a better illustration of our approach, we introduce a visualization of the permutation graph, called the *permutation diagram.* The permutation diagram consists of two horizontal parallel channels, named the top channel and the bottom channel. We put the numbers $1, 2, \ldots, n$ on the top channel, in this order, from left to right, and put the numbers $\pi[1], \pi[2], \ldots, \pi[n]$ on the bottom channel in the same way; then, for each $i \in V_n$, we draw a straight line joining two i's, one on the top channel and the other on the bottom channel. We label each such line by the same number i. Note that line i intersects line j iff i and j appear in reversed order in π. That is, the criterion for lines i and j to intersect is the same as for nodes i and j of the permutation graph to be adjacent. Therefore, an intersection graph of the lines of a permutation diagram is exactly the corresponding permutation graph.

Let D_1 and D_2 be subsets of V_n. (From now on, V_n denotes the set of lines of the permutation diagram and each member of V_n is called a *line*, with weight of line i equal to $w[i]$.) Then D_2 is said to *cover* D_1 if every line in $D_1 - D_2$ intersects some line in D_2. Our goal is to find a set of lines, say D, with minimum weight that covers the entire line set V_n of the permutation diagram. It is obvious that the set D then realizes a minimum-weight dominating set for G. For convenience, we first add two dummy nodes 0 and $n+1$, each with a weight 0, such that $\pi[0] = 0$ and $\pi[n+1] = n+1$, denoting the resulting graph by $G^+(\pi)$. Also, for any $j \geq 0$, V_j^+ denotes the set $\{0, 1, \ldots, j\}$. Note that any dominating set D^+ for G^+ directly realizes a dominating set for G by removing 0 and $n+1$ from D^+. Figure 1 shows the permutation diagram corresponding to $\pi = (2, 8, 6, 3, 1, 7, 4, 5)$ along with its weight function $w = (8, 3, 4, 1, 7, 1, 6, 12)$.

Any pair of lines i and j in V_{n+1}^+, $i \leq j$, is called an *ordered cross pair*, denoted by X_{ij}, if $i = j$ or i intersects (or crosses) j. For example, in Fig. 1, $(1, 2)$ and $(3, 3)$ are ordered cross pairs, while $(2, 3)$ and $(2, 1)$ are not. For any two ordered cross pairs X_{ij} and $X_{i'j'}$, X_{ij} is said to be *less* than $X_{i'j'}$, denoted by $X_{ij} < X_{i'j'}$, if (i) $j < j'$ or (ii) $j = j'$ and $i < i'$. For an ordered cross pair X_{ij}, we define $MWDS(X_{ij})$ to be a minimum-weight dominating set for V_j^+ such that $S_{ij} \subseteq MWDS(X_{ij})$, where

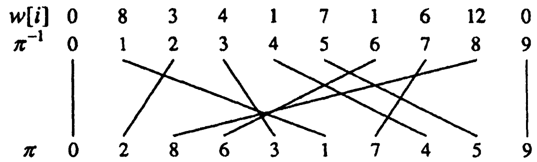


FIG. 1. *The permutation diagram for* $G^+(\pi)$.

$S_{ij} = \{i, j\}$. We first state a couple of results that appeared in [6].

THEOREM 2.1 (see [6]). *$MWDS(X_{(n+1)(n+1)})$ is a minimum-weight dominating set for G^+, and hence, $MWDS(X_{(n+1)(n+1)}) - \{0, n+1\}$ is a minimum-weight dominating set for G.*

THEOREM 2.2 (see [6]). *For each ordered cross pair $X_{ij}, j > 0$, there exists another cross pair $X_{i_bj_b}$, called back (X_{ij}), such that $X_{i_bj_b} < X_{ij}$ and $MWDS(X_{ij}) = MWDS(X_{i_bj_b}) + S_{ij}$.*

Observe that $back(X_{ij})$ is not unique in Theorem 2.2. That is, $back(X_{ij})$ is an arbitrary cross pair satisfying the given conditions. Throughout the paper, for the purpose of convenience, when a notation, such as $back(X_{ij})$, is introduced to indicate a single object, rather than a set, then an arbitrary one will be selected if more than one candidate is available. Theorem 2.2 leads to a natural algorithm for computing $MWDS(X_{(n+1)(n+1)})$ using the so-called dynamic programming technique similar to the one used in [6]. Once all $back(X_{ij})$'s are computed, $MWDS(X_{(n+1)(n+1)})$ is easily obtained by tracing through the relation *back*. That is, $MWDS(X_{(n+1)(n+1)}) = S_{i_{k+1}j_{k+1}} + S_{i_kj_k} + \cdots + S_{i_0j_0}$, where $X_{i_{k+1}j_{k+1}} = X_{(n+1)(n+1)}$, $X_{i_0j_0} = X_{00}$, and $back(X_{i_sj_s}) = X_{i_{s-1}j_{s-1}}$ for $s = k+1, k, \ldots, 1$. The algorithm in [6] consists of n stages and at the jth stage it finds an $MWDS(X_{ij})$ for each ordered cross pair X_{ij}, $i \leq j$. Each of these stages takes $O(m+n)$ steps, giving an overall time complexity of $O(n(m+n))$. The algorithm in this paper basically employs a similar technique but exploits some of the properties of permutation graphs to avoid duplicate computations so that the total time complexity is sharply reduced to $O(m+n)$.

3. The algorithm.

Before stating the algorithm, we first introduce a few definitions. We define four crossing lists, *TLCL* (*Top_Left Crossing List*), *TRCL* (*Top_Right Crossing List*), *BLCL* (*Bottom_Left Crossing List*), and *BRCL* (*Bottom_Right Crossing List*), that are used in computing $back(X_{ij})$. For each $i \in V^+_{n+1}$, $TLCL_i$ denotes the ordered list $(k_0, k_1, \ldots, k_r)$ such that each k_t, $0 \leq t < r$, crosses i and $k_0 < k_1 < \cdots < k_r = i$. That is, $TLCL_i$ is the list of the lines, maintained in the increasing order of their numbers, crossing i from the left side of i in the top channel of the permutation diagram to the right side of i in the bottom channel. Note that i itself is a member of the list. The rest of the lists are defined as follows.

$TRCL_i$ is the ordered list $(k_0, k_1, \ldots, k_r)$ such that each k_t, $0 < t \leq r$, crosses i and $i = k_0 < k_1 < \cdots < k_r$.

$BLCL_i$ is the ordered list $(k_0, k_1, \ldots, k_r)$ such that each k_t, $0 \leq t < r$, crosses i and $\pi^{-1}[k_0] < \pi^{-1}[k_1] < \cdots < \pi^{-1}[k_r] = \pi^{-1}[i]$.

$BRCL_i$ is the ordered list $(k_0, k_1, \ldots, k_r)$ such that each k_t, $0 < t \leq r$, crosses i and $\pi^{-1}[i] = \pi^{-1}[k_0] < \pi^{-1}[k_1] < \cdots < \pi^{-1}[k_r]$.

In Fig. 1, we have $TLCL_2 = (1,2)$, $TLCL_8 = (1,3,4,5,6,7,8)$, $TRCL_1 = (1,2,3,6,8)$, $TRCL_2 = (2)$, $BLCL_1 = (2,8,6,3,1)$, $BLCL_8 = (8)$, $BRCL_2 = (2,1)$, and $BRCL_8 = (8,6,3,1,7,4,5)$. Note that $TLCL_i$ and $BRCL_i$ contain the same set of lines in a different order. The same is true for $BLCL_i$ and $TRCL_i$. Lemma 4.1 describes an efficient method for computing the various crossing lists defined above.

We now concentrate on the computation of $back(X_{ij})$ along with $mw(X_{ij})$ for each X_{ij}, where $mw(X_{ij})$ denotes the weight of $MWDS(X_{ij})$. Let $X_{i_b j_b}$ denote a $back(X_{ij})$. As shown in the proof of Theorem 2.2 [6], $back(X_{ij})$ is an ordered cross pair with the smallest mw value among all the pairs that satisfy one of the following conditions:

(1) $i_b = i$ and $i < j_b < j$;

(2) $i_b < i$ and $j_b = j$; or

(3) $j_b < i$, $\pi^{-1}[j_b] < \pi^{-1}[j]$, and the set of lines between j_b and i is covered by $S_{i_b j_b} + S_{ij}$.

Note that i_b may intersect i or j or both. The three cases are shown pictorially in Fig. 2. For convenience, we let $iback(X_{ij})$, $jback(X_{ij})$, and $xback(X_{ij})$ indicate the $back(X_{ij})$ satisfying conditions (1), (2), and (3), respectively. For any X_{ij}, $back(X_{ij})$ is then one of $iback(X_{ij})$, $jback(X_{ij})$, and $xback(X_{ij})$, whichever results in the smallest $mw(X_{ij})$ value, i.e., the smallest among $W(MWDS(iback(X_{ij})) + S_{ij})$, $W(MWDS(jback(X_{ij})) + S_{ij})$, and $W(MWDS(xback(X_{ij})) + S_{ij})$. Note that at least one of the three different *backs* exists for each X_{ij} except for X_{00}. Figure 2 illustrates these different types of $back(X_{ij})$. For example, in Fig. 1, X_{56} and X_{57} are candidates for $iback(X_{58})$. Since $mw(X_{57}) = 17$ and $mw(X_{56}) = 11$, we let $iback(X_{58}) = X_{56}$. Among X_{48}, X_{38}, and X_{18}, which are candidates for $jback(X_{58})$, we pick X_{48} since $mw(X_{48}) = 16$, whereas $mw(X_{18}) = 20$ and $mw(X_{38}) = 19$. The candidates for $xback(X_{58})$ are X_{12} and X_{22}. Therefore, $xback(X_{58}) = X_{22}$, since $mw(X_{22}) = 3$ is smaller than $mw(X_{12}) = 11$. Thus in this case $back(X_{58}) = X_{22}$ because $xback(X_{58})$ results in the smallest $mw(X_{58})$. Note that there are no candidates for $iback(X_{36})$ and $jback(X_{11})$. The procedures for computing $iback(X_{ij})$ and $jback(X_{ij})$ are relatively simple and are discussed in Lemma 4.4. In the following, we show how to identify $xback(X_{ij})$.

We first introduce a few more definitions. For each X_{ij}, we define the *clique* of X_{ij}, denoted by $CLIQ(X_{ij})$, to be the set of lines, say k, satisfying $i \le k \le j$, $\pi^{-1}[j] \le \pi^{-1}[k] \le \pi^{-1}[i]$ and there is no line k' such that $k < k' < j$ and $\pi^{-1}[k] < \pi^{-1}[k'] < \pi^{-1}[i]$. That is, $CLIQ(X_{ij})$ is the set of lines between i and j, including i and j, that form a clique, and if two lines k and k', $k < k'$, are in two such different cliques, then we choose the clique that contains the line k', as illustrated in Fig. 3. For example, in Fig. 1, $CLIQ(X_{18}) = \{1, 3, 6, 8\}$, $CLIQ(X_{48}) = \{4, 7, 8\}$, $CLIQ(X_{13}) = \{1, 3\}$, and $CLIQ(X_{11}) = \{1\}$.

For each X_{ij}, we define a *minimum cover* of X_{ij}, denoted $mc(X_{ij})$, to be an ordered cross pair $X_{i_m j_m}$ such that $mw(X_{i_m j_m})$ is minimum over all pairs satisfying the following two conditions:

(1) $i \le j_m \le j$, $\pi^{-1}[j_m] \le \pi^{-1}[i]$, and

(2) $S_{i_m j_m}$ covers $CLIQ(X_{ij})$.

Note that the existence of $mc(X_{ij})$ for each X_{ij} is assured since for any $k \in CLIQ(X_{ij})$, X_{kk} satisfies (1) and (2) and, therefore, is a candidate for $mc(X_{ij})$. In Fig. 1, we have $mc(X_{18}) = X_{66}$ since $mw(X_{11}) = 8$, $mw(X_{12}) = 11$, $mw(X_{13}) = 12$, $mw(X_{16}) = 9$, $mw(X_{18}) = 20$, $mw(X_{33}) = 7$, $mw(X_{36}) = 8$, $mw(X_{38}) = 19$, $mw(X_{46}) = 5$, $mw(X_{48}) = 16$, $mw(X_{56}) = 11$, $mw(X_{58}) = 22$, $mw(X_{66}) = 4$, $mw(X_{68}) = 16$, $mw(X_{78}) = 21$, and $mw(X_{88}) = 15$. Similarly, $mc(X_{12}) = X_{22}$ and $mc(X_{57}) = X_{46}$.

For each X_{ij}, except X_{00}, let $V_{ij} = \{k \mid k < i, \pi^{-1}[k] < \pi^{-1}[j]\}$. That is, V_{ij} is

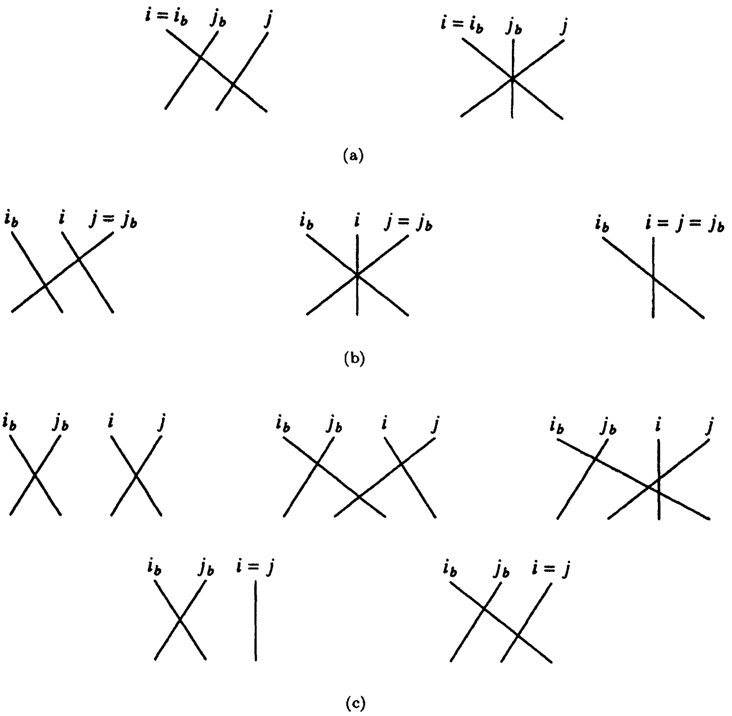


FIG. 2. *Illustration for* $iback(X_{ij})$, $jback(X_{ij})$, $xback(X_{ij})$. (a) $i_b = i$ *and* $i < j_b < j$; (b) $i_b < i$ *and* $j_b = j$; (c) $j_b < i$ *and* $\pi^{-1}[j_b] < \pi^{-1}[j]$.

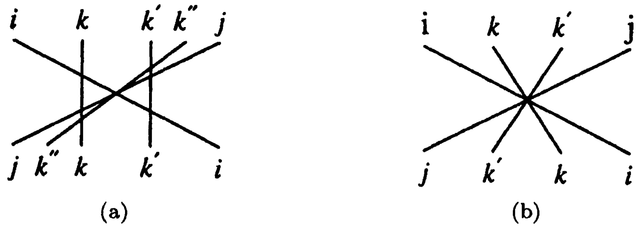


FIG. 3. *Examples of* $CLIQ(X_{ij})$. (a) $CLIQ(X_{ij}) = \{i, k', k'', j\}$; (b) $CLIQ(X_{ij}) = \{i, k, k', j\}$.

the set of lines less than i that do not intersect the line j. Then $limit(X_{ij})$ is the ordered cross pair, say $X_{i_l j_l}$, such that (i) $i_l, j_l \in V_{ij}$ and (ii) j_l is the largest number in V_{ij} and i_l is such that for all $k \in V_{ij}$, $\pi^{-1}[i_l] \geq \pi^{-1}[k]$. For example, in Fig. 1, $limit(X_{11}) = X_{00}$, $limit(X_{46}) = X_{22}$, $limit(X_{57}) = X_{13}$, and $limit(X_{99}) = X_{58}$. It is readily seen that for each X_{ij}, there is a unique $limit(X_{ij})$. As will be seen shortly, $limit(X_{ij})$ is useful in cutting down the amount of work needed to compute $xback(X_{ij})$. The computation of $limit(X_{ij})$ is discussed in Lemma 4.2.

We now return to the computation of $xback(X_{ij})$. By definition, $xback(X_{ij})$ is a cross pair, say $X_{i_b j_b}$, with the smallest *mw* value among the pairs that satisfies the following two conditions:

(1) $j_b < i$, $\pi^{-1}[j_b] < \pi^{-1}[j]$, and

(2) the set of lines k such that $j_b < k < i$ is covered by $S_{i_b j_b} + S_{ij}$.

A brute force method to find such a pair would be to check every pair that satisfies condition (1) to see whether it also satisfies condition (2). We, however, can put a limit in searching such pairs with the help of $limit(X_{ij})$.

THEOREM 3.1. *For any $X_{ij} \neq X_{00}$, let $X_{i_l j_l} = limit(X_{ij})$ and $X_{i_m j_m} = mc(X_{i_l j_l})$. Then $mw(xback(X_{ij})) = mw(X_{i_m j_m})$, if $W(MWDS(xback(X_{ij})) + S_{ij}) < W(MWDS(jback(X_{ij})) + S_{ij})$.*

Proof. Let $S_1 = \{X_{i'j'} | i_l \leq j' \leq j_l,\ \pi^{-1}[j'] \leq \pi^{-1}[i_l]$, and $S_{i'j'}$ covers $CLIQ(X_{i_l j_l})\}$ and $S_2 = \{X_{i'j'} | j' < i,\ \pi^{-1}[j'] < \pi^{-1}[j]$, and for any k such that $j' < k < i$, $\{k\}$ is covered by $S_{i'j'} + S_{ij}\}$. By definition, $X_{i_m j_m}$ and $xback(X_{ij})$ are the cross pairs in S_1 and S_2, respectively, with the smallest mw value. We show that $S_1 \subseteq S_2$ and the mw value of any cross pair in $S_2 - S_1$ is no less than the smallest mw value in S_1. First, prove $S_1 \subseteq S_2$. Let $X_{i'j'} \in S_1$. Then clearly $j' < i$ and $\pi^{-1}[j'] < \pi^{-1}[j]$ since $j_l < i$ and $\pi^{-1}[i_l] < \pi^{-1}[j]$. Next, let k be a line such that $j' < k < i$. If $j_l < k$, then S_{ij} covers $\{k\}$ by the definition of $limit(X_{ij})$. If $j' < k < j_l$ and $k \notin CLIQ(X_{i_l j_l})$, then we must have one of the following three cases:

(1) $\pi^{-1}[k] > \pi^{-1}[j]$,

(2) $\pi^{-1}[k] < \pi^{-1}[j_l]$, or

(3) there is a line $r \in CLIQ(X_{i_l j_l})$, such that $k < r$, and $\pi^{-1}[k] < \pi^{-1}[r]$.

In case (1), $\{k\}$ is covered by $\{j\}$, hence by S_{ij}. In case (2), it is covered by $\{i'\}$, hence by $S_{i'j'}$. In case (3), $\{r\}$ is covered by $S_{i'j'}$, so is $\{k\}$. Combining all these, we get $S_1 \subseteq S_2$.

Next, we show that the mw value of any cross pair in $S_2 - S_1$ is no less than the smallest mw value in S_1. Suppose it is not true, and let $X_{i'j'}$ be a cross pair with the smallest mw value in $S_2 - S_1$. That is, $mw(xback(X_{ij})) = mw(X_{i'j'})$. Since $X_{i'j'} \in S_2 - S_1$, we have $j' < i_l$, which implies $\pi^{-1}[i'] > \pi^{-1}[i_l]$ in order to cover $\{i_l\}$. This forces $X_{i'j}$ to be a candidate for $jback(X_{ij})$. Then $W(MWDS(jback(X_{ij}) + S_{ij})) \leq W(MWDS(X_{i'j'}) + S_{ij})) = W(MWDS(xback(X_{ij}) + S_{ij}))$. This contradicts the condition that $W(MWDS(xback(X_{ij}) + S_{ij})) < W(MWDS(jback(X_{ij}) + S_{ij}))$. □

The significance of this theorem is that instead of explicitly finding $xback(X_{ij})$ using its definition, we simply make use of $mc(limit(X_{ij}))$, which is already available as a result of the computation in the previous stages.

Now we show how to find $mc(X_{ij})$ for each X_{ij}. To help us identify such a pair, we use a line in $CLIQ(X_{ij})$ that divides $CLIQ(X_{ij})$ into two parts. This line, denoted by $cdiv(X_{ij})$, is defined as

$$cdiv(X_{ij}) = \begin{cases} i & \text{if } i = j, \\ \max\{CLIQ(X_{ij}) - \{j\}\} & \text{if } i \neq j. \end{cases}$$

Now consider the following two cases.

Case 1. $cdiv(X_{ij}) \neq i$. Let $cdiv(X_{ij}) = k$. Then it is not difficult to show that k divides $CLIQ(X_{ij})$ into two parts: $CLIQ(X_{ik})$ and $CLIQ(X_{kj})$. In Theorem 3.2, we show that $mc(X_{ij})$ is either $mc(X_{ik})$ or $mc(X_{kj})$, whichever has a smaller mw value.

Case 2. $cdiv(X_{ij}) = i$. Note that in this case, either $CLIQ(X_{ij}) = \{i\}$ if $i = j$ or $CLIQ(X_{ij}) = \{i, j\}$ if $i \neq j$. We compute $mc(X_{ij})$ from scratch. For convenience, let $mc2(X_{ij})$ denote $mc(X_{ij})$ when $|CLIQ(X_{ij})| \leq 2$. In other words, $mc2(X_{ij})$ is an ordered cross pair with the smallest mw value among all pairs, say $X_{i'j'}$, satisfying the following conditions:

(1) $cdiv(X_{ij}) = i$,

(2) $i \leq j' \leq j$, $\pi^{-1}[j'] \leq \pi^{-1}[i]$, and

(3) $S_{i'j'}$ covers $CLIQ(X_{ij})$;

Section 5 is devoted to computing all $mc2(X_{ij})$ efficiently.

THEOREM 3.2.

(a) *Suppose* $cdiv(X_{ij}) = k(\neq i)$. *Then* $mc(X_{ij}) = X_{i_m j_m} \in \{mc(X_{ik}), mc(X_{kj})\}$ *such that* $mw(X_{i_m j_m})$ *is a smaller of* $mw(mc(X_{ik}))$ *and* $mw(mc(X_{kj}))$.

(b) *Suppose* $cdiv(X_{ij}) = i$. *Then* $mc(X_{ij}) = mc2(X_{ij})$.

Proof. (a) Let

$C = \{X_{i'j'} \,|i \le j' \le j, \pi^{-1}[j'] \le \pi^{-1}[i]$, and $S_{i'j'}$ covers $CLIQ(X_{ij})\}$,

$C_1 = \{X_{i'j'} \,|i \le j' \le k, \pi^{-1}[j'] \le \pi^{-1}[i]$, and $S_{i'j'}$ covers $CLIQ(X_{ik})\}$, and

$C_2 = \{X_{i'j'} \,|k \le j' \le j, \pi^{-1}[j'] \le \pi^{-1}[k]$, and $S_{i'j'}$ covers $CLIQ(X_{kj})\}$.

Then it suffices to prove that $C = C_1 + C_2$. To prove $C_1 + C_2 \subseteq C$, first let $X_{i'j'} \in C_1$. Consider two cases: either (i) i' or j' is a member of $CLIQ(X_{ik})$, or (ii) neither of them is a member of $CLIQ(X_{ik})$. In (i), obviously $\{i'\}$ (or $\{j'\}$) covers $CLIQ(X_{ij})$. In (ii), since $j' < k$ and hence j' does not cross k, i' should cross k in such a way that $i' < k < j$ and $\pi^{-1}[j] < \pi^{-1}[k] < \pi^{-1}[i']$, which means that i' also crosses j. Therefore, $S_{i'j'}$ covers $CLIQ(X_{ij})$. In both cases, we have $X_{i'j'} \in C$. The other case, i.e., when $X_{i'j'} \in C_2$, can be proved similarly.

Now, to prove that $C \subseteq C_1 + C_2$, let $X_{i'j'} \in C$. Again consider two cases: (i) If i' or j' is a member of $CLIQ(X_{ij})$, then clearly $S_{i'j'}$ covers $CLIQ(X_{ij})$ and $CLIQ(X_{kj})$. Therefore, $X_{i'j'} \in C_1$ if $i \le j' \le k$, or $X_{i'j'} \in C_2$ if $k \le j' \le j$. (ii) If neither i' nor j' is a member of $CLIQ(X_{ij})$, then check two subcases: $k < j' < j$ or $i < j' < k$. If $k < j' < j$, then $\pi^{-1}[j'] < \pi^{-1}[j] < \pi^{-1}[k]$, which implies that j' intersects k but not j. This in turn implies that i' crosses j, hence, $S_{ij'}$ covers $CLIQ(X_{kj})$. Therefore, $X_{i'j'} \in C_2$. Suppose $i < j' < k$. Since j' is not a member of $CLIQ(X_{ij})$ and $\pi^{-1}[j'] < \pi^{-1}[i]$, we have $\pi^{-1}[j'] < \pi^{-1}[k]$, which implies that $\{k\}$ is to be covered by $\{i'\}$. Then $\pi^{-1}[k] < \pi^{-1}[i']$. Hence $CLIQ(X_{ik})$ is covered by $S_{i'j'}$.

(b) Part (b) of the theorem follows directly from the definition. □

Before closing this section, we outline the algorithm for computing $back(X_{ij})$ in Algorithm 3.1.

```
ALGORITHM 3.1. Computing $back(X_{ij})$ for all $X_{ij}$'s
{initialization}
$mw(X_{00}) = 0$;
$mc(X_{00}) = X_{00}$;
{See §4 for the computation of these functions.}
compute $TLCL_j$, $TRCL_j$, $BLCL_j$, and $BRCL_j$ for each $j$;
compute $limit(X_{ij})$ and $cdiv(X_{ij})$ for each $X_{ij}$;

for each $j = 1, 2, \ldots, n+1$ do
    {compute back $(X_{ij})$ along with $mw(X_{ij})$.}
    for each $i \in TLCL_j$ do in the increasing order of $i$
        compute $iback(X_{ij})$, $jback(X_{ij})$, and $xback(X_{ij})$;
        {See Lemmas 4.4 and 4.2.}
        $back(X_{ij}) \leftarrow$ an ordered cross pair with minimum value among
                          $W(MWDS(iback(X_{ij})) + S_{ij})$,
                          $W(MWDS(jback(X_{ij})) + S_{ij})$, and
                          $W(MWDS(xback(X_{ij})) + S_{ij})$;

                            { $W(MWDS(iback(X_{ij})) + S_{ij})$,
                            $W(MWDS(jback(X_{ij})) + S_{ij})$, and
                               $W(MWDS(xback(X_{ij})) + S_{ij})$
```

```
                    were obtained in previous stages.}

        update $mw(X_{ij})$
    endfor; {$i$}

    {compute $mc2(X_{ij})$ for each $i \in TLCL_j$ with $|CLIQ(X_{ij})| \leq 2$.}
    call $find\_mc2(j)$; {See §5.}

    {compute $mc(X_{ij})$ for each $i \in TLCL_j$. The $mc(X_{ij})$'s computed in this stage
            will be used to identify $xback$ values in later stages.}
    for each $i \in TLCL_j$ do in decreasing order of $i$
        if $cdiv(X_{ij}) = k(\neq i)$
            then
                $mc(X_{ij}) \leftarrow$ either $mc(X_{ik})$ or $mc(X_{kj})$,
                    whichever has a smaller $mw$ value;
            else
                $mc(X_{ij}) \leftarrow mc2(X_{ij})$
    endfor; {$i$}
endfor; {$j$}
```

4. Subroutines. In this section, we show how to compute the four lists (*TLCL*, *TRCL*, *BLCL*, and *BRCL*) and basic functions such as *limit*, *cdiv*, *iback*, and *jback* efficiently.

LEMMA 4.1. *All $TLCL_j$'s, $TRCL_j$'s, $BLCL_j$'s, and $BRCL_j$'s can be computed in $O(m+n)$ time.*

Proof. To compute $TLCL_j$ for all $j \in V_{n+1}^{+}$, we inspect the given permutation $\pi = (\pi[1], \pi[2], \ldots, \pi[n+1])$ in its reverse order from $\pi[n+1]$ to $\pi[1]$. Let j be an element of the permutation in this order. We traverse a sorted list L, from the beginning, comparing each element of L with j until we insert j at the proper place in L, where L is maintained in the increasing order of its elements. Initially L is empty and $j = \pi[n+1]$. When j is inserted into L, the sequence $k_0, k_1, \ldots, k_{r-1}$ of elements in L that are less than j, along with j itself, forms $TLCL_j$, since $k_0 < k_1 < \cdots < k_{r-1} < j$ and $\pi^{-1}[k_0] > \pi^{-1}[k_1] > \cdots > \pi^{-1}[k_{r-1}] > \pi^{-1}[j]$. For example, let $\pi = (4\ 1\ 3\ 2\ 5)$. Initially L is empty. We first add 5 to L. In this case, no comparison is required and we have $TLCL_5 = (5)$. Next, add 2 to the front of 5 since 2 is less than 5, making $L = \{2, 5\}$. Since there is no element less than 2 in L at this time, we have $TLCL_2 = (2)$. For $j = 3$, put 3 between 2 and 5, making $L = \{2, 3, 5\}$, which gives (2, 3) as $TLCL_3$. For $j = 1$, add j to the beginning of L and let $TLCL_1 = (1)$. Finally for $j = 4$, we put j right after 3, making $L = \{1, 2, 3, 4, 5\}$, and get $TLCL_4 = (1, 2, 3, 4)$. Note that the number of comparisons made at the time of insertion of j into L is equal to the number of elements in $TLCL_j$. Thus, the time required in the above procedure is $\sum_{j=1}^{n+1} |TLCL_j| = O(m+n)$ steps, where $|TLCL_j|$ denotes the length of $TLCL_j$.

The remaining lists can be obtained in a similar way each with a time complexity of $O(m+n)$. □

LEMMA 4.2. *All limit (X_{ij})'s can be computed in time $O(m+n)$.*

Proof. Let $X_{i_l j_l} = limit(X_{ij})$, $limit(X_{ij}).i = i_l$, and $limit(X_{ij}).j = j_l$. We first compute $limit(X_{ij}).j$ for each $i \in TLCL_j$. For each j, $1 \leq j \leq n+1$, we consider $TLCL_j = (i_0, i_1, \ldots, i_r = j)$. Since line $i_0 - 1$ does not cross j, we have $limit(X_{i_0 j}).j = i_0 - 1$. Next, to compute $limit(X_{i_1 j}).j$, we compare i_0 and i_1. If their difference, $i_1 - i_0$, is only 1, then $limit(X_{i_1j}).j$ is again $i_0 - 1$, since there is no line between i_0 and i_1. If their difference is greater than 1, then $i_1 - 1$ should be $limit(X_{i_1j}).j$, because $i_1 - 1$ is the largest line that crosses neither i nor j. We continue the same computation for each $limit(X_{i_sj}).j$, $s = 2, \ldots, r$, setting $limit(X_{i_sj}).j$ to be $limit(X_{i_{s-1}j}).j$ if $i_s - i_{s-1} = 1$ and to $i_s - 1$ otherwise. The time required is exactly $m + n$ steps, since each element of $TLCL_j$ is inspected only once.

Next, we compute $limit(X_{ij}).i$ in a similar way. For each i, $1 \leq i \leq n+1$, we consider $BLCL_i = (j_0, j_1, \ldots, j_r = i)$. Let $k = \pi[\pi^{-1}[j_0] - 1]$. Since k does not cross i or j, we have $limit(X_{ij_0}).i = k$. To compute $limit(X_{ij_1}).i$, we compare $\pi^{-1}[j_0]$ and $\pi^{-1}[j_1]$. If their difference, $\pi^{-1}[j_1] - \pi^{-1}[j_0]$, is only 1, $limit(X_{ij_1}).i$ is again k, since $\pi^{-1}[k]$ is the largest among all $\pi^{-1}[k']$ such that $k' < i$ and $\pi^{-1}[k'] < \pi^{-1}[j_1]$. If their difference is greater than 1, then $\pi[\pi^{-1}[j_1] - 1]$ should be $limit(X_{ij_1}).i$. We continue the same computation for each $limit(X_{ij_s}).i$, $s = 2, \ldots, r$. Again the time required is exactly $m + n$, since each element of $BLCL_i$ is inspected only once. □

LEMMA 4.3. *All* $cdiv(X_{ij})$*'s can be computed in* $O(m+n)$ *time.*

Proof. For each $j \in V^+_{n+1}$, we traverse $TLCL_j = (i_0, i_1, \ldots, i_r = j)$ and $BRCL_j = (k_0 = j, k_1, \ldots, k_r)$ in their reverse order, from right to left. Obviously, $cdiv(X_{jj}) = j$. We first consider i_{r-1}, which is the second largest line in $TLCL_j$. For this i_{r-1}, we visit each member of $BRCL_j$ in its reverse order starting at k_r to identify every pair X_{k_sj}, $s = r, r-1, \ldots, 1$, whose $cdiv$ is i_{r-1}. It is straightforward to see that i_{r-1} is $cdiv$ for each pair X_{k_sj} such that k_s is a member of $BRCL_j$ satisfying $\pi^{-1}[k_s] \geq \pi^{-1}[i_{r-1}]$. Let k_t be the first member of $BRCL_j$, going from right to left, such that $\pi^{-1}[k_t] \leq \pi^{-1}[i_{r-1}]$. Since i_{r-1} cannot be $cdiv(X_{k_tj})$ any more, consider i_{r-2} next. Again, for this i_{r-2}, we traverse $BRCL_j$ again in reverse order starting from k_t and setting $cdiv(X_{k_tj})$ to i_{r-2} for each k_s in $BRCL_j$ such that $\pi^{-1}[k_s] \geq \pi^{-1}[i_{r-2}]$. We repeat the above procedure until we get $cdiv(X_{ij})$'s for all i's crossing j. As an example, consider Fig. 1. Let $j = 8$. Then $TLCL_8 = (1\ 3\ 4\ 5\ 6\ 7\ 8)$ and $BRCL_8 = (8\ 6\ 3\ 1\ 7\ 4\ 5)$. Then we set $cdiv(X_{88})$ to 8. Next, let 7 be $cdiv$ for X_{58}, X_{48}, and X_{78} since $\pi^{-1}[5] \geq \pi^{-1}[7]$, $\pi^{-1}[4] \geq \pi^{-1}[7]$, and $\pi^{-1}[7] \geq \pi^{-1}[7]$. Next, we consider 6 in $TLCL_8$ and traverse $BRCL_8$ starting at 1 from right to left and let 6 be $cdiv$ for X_{18}, X_{38}, and X_{68} for the same reason. This procedure takes $|TLCL_j| + |BRCL_j|$ steps for each j. Therefore, the time complexity for computing all $cdiv(X_{ij})$'s is $O(m+n)$. □

LEMMA 4.4. *All* $iback(X_{ij})$*'s and* $jback(X_{ij})$*'s can be computed in* $O(m+n)$ *time.*

Proof. We first show how to compute all $iback(X_{ij})$. Let $TRCL_i = \{i = j_0, \ldots, j_r\}$ with $j = j_t$ for some $t \in \{0, \ldots, r\}$. Note that $iback(X_{ij_0})$ $(= iback(X_{ii}))$ and $iback(X_{ij_1})$ are not defined.

Note that $iback(X_{ij_2}) = X_{ij_1}$ because X_{ij_1} is the only candidate. For $k = 3, \ldots, r$, $iback(X_{ij_k})$ is either $X_{ij_{k-1}}$ or $iback(X_{ij_{k-1}})$, whichever has a smaller mw value. The above operation is basically the prefix function on mw values of $X_{ij_1}, X_{ij_2}, \ldots$, and X_{ij_r}. Note that, in Algorithm 3.1, the $mw(X_{ij})$'s are computed in the increasing order of j. Since the total number of $iback(X_{ij})$'s is $\leq \sum_{i=1}^{n+1} |TRCL_i| \leq m+n$, the result follows.

Next, to compute all $jback(X_{ij})$, we perform a similar prefix computation using $TLCL_j$ instead of $TRCL_i$. □

5. Computing $mc2(X_{ij})$. According to its definition, $mc2(X_{ij})$ is to be computed only if $|CLIQ(X_{ij})| \leq 2$. If $|CLIQ(X_{ij})| = 1$, i.e., $i = j$, then $mc2(X_{ij}) = X_{jj}$ because $mw(X_{jj}) \leq mw(X_{i'j})$ for any cross pair $X_{i'j}$. In the following, therefore, we

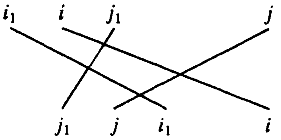


FIG. 4. *Illustration of* $base(X_{ij})$.

consider only the cross pairs X_{ij} such that $|CLIQ(X_{ij})| = 2$. We first introduce some notation that helps characterize all candidates for $mc2(X_{ij})$. To compute $mc2(X_{ij})$ when $|CLIQ(X_{ij})| = 2$, we need to consider all the ordered cross pairs $X_{i'j'}$ such that $i \leq j' \leq j$, $\pi^{-1}[j'] \leq \pi^{-1}[i]$, and $S_{i'j'}$ covers S_{ij} and then choose a pair for which the mw value is a minimum. Obviously, X_{ii} and X_{jj} are candidates for $mc2(X_{ij})$. Below we identify all candidates for $mc2(X_{ij})$ other than X_{ii} and X_{jj}.

5.1. Identifying candidates for $mc2(X_{ij})$. Let l be any line in V_{n+1}^{+} such that $|TRCL_l| \geq 2$. Define $j\_base[l]$ to be the second member of $TRCL_l$. Similarly, $i\_base[l]$ is the second member of $BRCL_l$, provided $|BRCL_l| \geq 2$. Now let X_{ij} be a cross pair such that $|CLIQ(X_{ij})| = 2$. Then $|TRCL_i| \geq 2$ and $|BRCL_j| \geq 2$. Let $i_1 = i\_base[j]$ and $j_1 = j\_base[i]$. Then it is easily observable that $i_1 \neq j_1$ and they cross each other as shown in Fig. 4. This ordered cross pair $X_{i_1 j_1}$ is called the *base pair* of X_{ij}, denoted by $base(X_{ij})$. It is obvious that such a base pair always exists for any given X_{ij} such that $|CLIQ(X_{ij})| = 2$ since X_{ij} itself may be its base pair.

Let $X_{i_1 j_1}$ be $base(X_{ij})$. Then let $TRCL_{i_1}[j_1, j] = (j_1, j_2, \dots, j_r = j)$ be the sublist of $TRCL_{i_1}$ consisting of all elements of $TRCL_{i_1}$ from j_1 up to j, preserving the order in $TRCL_{i_1}$. Further, for each such j_k, $1 \leq k \leq r$, in $TRCL_{i_1}[j_1, j]$, let $BRCL_{j_k}[i_1] = (i_{f_k(1)} = i_1, i_{f_k(2)}, \dots, i_{f_k(s_k)})$, be the sublist of $BRCL_{j_k}$ consisting of all elements from i_1 to the last member of $BRCL_{j_k}$, keeping the order intact. Define

$$\begin{aligned} C(X_{ij}) &= \{X_{i'j'} | X_{i_1 j_1} = base(X_{ij}) \text{ and } j' \in TRCL_{i_1}[j_1, j], i' \in BRCL_{j'}[i_1]\} \\ &= \{X_{i_{f_1(1)} j_1}, X_{i_{f_1(2)} j_1}, \dots, X_{i_{f_1(s_1)} j_1}, X_{i_{f_2(2)} j_2}, \dots, \\ &\qquad X_{i_{f_2(s_2)} j_2}, \dots, X_{i_{f_r(1)} j_r}, X_{i_{f_r(2)} j_r}, \dots, X_{i_{f_r(s_r)} j_r}\}. \end{aligned}$$

Then $C(X_{ij})$ consists of all candidates for $mc2(X_{ij})$ (other than X_{ii} and X_{jj}).

To choose $mc2(X_{ij})$ from $C(X_{ij})$ efficiently, we introduce an additional definition. Let $X_{i_1 j_1}$ be $base(X_{ij})$, and let $TRCL_{i_1}[j_1, j] = (j_1, j_2, \dots, j_r = j)$ and $BRCL_{j_k}[i_1] = (i_{f_k(1)} = i_1, i_{f_k(2)}, \dots, i_{f_k(s_k)})$ be the sublists defined above. For each k, $1 \leq k \leq r$, define $minv(X_{i_1 j_k})$ to be the minimum of $mw(X_{i_1 j_k})$, $mw(X_{i_{f_k(2)} j_k})$, $\dots$, and $mw(X_{i_{f_k(s_k)} j_k})$. Then to find $mc2(X_{ij})$ in $C(X_{ij})$, compare each of $minv(X_{i_1 j_1})$, $minv(X_{i_1 j_2}), \dots$, and $minv(X_{i_1 j_r})$, and pick a cross pair with the smallest $minv$ value. That is, for computing $mw(mc2(X_{ij}))$, we just make use of $minv(X_{i_1 j_k})$ values for all $j_k \in TRCL_{i_1}[j_1, j]$. It is not difficult to maintain these $minv$'s so that they may be available when we search for $mc2(X_{ij})$. After computing $mw(X_{ij})$'s, for all $i \in TLCL_j$ in Algorithm 3.1, apply, for example, the prefix computation on the mw values. To be specific, let $BRCL_j = (i_0 = j, i_1, \dots, i_r)$. Then $minv(X_{i_r j}) = mw(X_{i_r j})$, and for $s = r-1, r-2, \dots, 1$, $minv(X_{i_s j}) = \min\{mw(X_{i_s j}), minv(X_{i_{s+1} j})\}$. Obviously, for each j, it requires $|BRCL_j|$ time to compute all $minv(X_{ij})$'s, and thus the overall time to compute all $minv$'s is $\sum_{j=1}^{n} |BRCL_j| = O(m + n)$.

The use of $minv$ values considerably reduces the total number of comparisons needed in computing $mc2(X_{ij})$ for each X_{ij}. However, this reduction is not sufficient

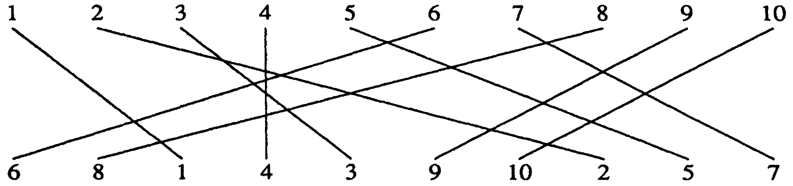


FIG. 5. *A sample graph for computing* $mc2(X_{ij})$.

to meet our goal since there are still potentially n values to be compared in computing each $mc2(X_{ij})$. Fortunately, $C(X_{ij})$ is well behaved, and it is possible to organize the computation of all $mc2(X_{ij})$'s in an orderly fashion to take advantage of the comparisons already made. To keep the total number of comparisons within desirable limits, we access each $minv(X_{ij})$ at most once during the entire run of our algorithm. The following observations are not hard to prove.

(1) $C(X_{ij'}) \subseteq C(X_{ij})$ if (i) $j' \leq j$ and (ii) $i\_base[j'] = i\_base[j]$.

(2) $C(X_{i'j}) \subseteq C(X_{ij})$ for $i \leq i'$.

The above properties suggest computing each $mw(mc2(X_{ij}))$ in increasing order of j and then in decreasing order of i for each $i \in TLCL_j$. Therefore, as we go from j' to j, $j' < j$, $mc2(X_{ij'})$'s are available for all $i \in TLCL_{j'}$ such that $|CLIQ(X_{ij'})| \leq 2$. Also, while at j, if $i' > i$, then $mc2(X_{i'j})$ is determined ahead of $mc2(X_{ij})$.

5.2. The algorithm for computing $mc2(X_{ij})$. Now we develop a formal procedure for all $mc2(X_{ij})$'s. At the jth stage, $0 \leq j \leq n+1$, visit each $i \in TLCL_j$ in reversed order starting at j. Note that we consider only i such that $|CLIQ(X_{ij})| = 2$. Let RS_TLCL_j denote the reversed sublist of $TLCL_j$ consisting of all such i's. Note that no two lines in RS_TLCL_j cross each other and i_base$[j]$ is also in RS_TLCL_j as the last member. Since $|CLIQ(X_{ij})| = 2$ only if $cdiv(X_{ij}) = i \neq j$, and since all $cdiv(X_{ij})$'s are known in advance, it is easy to determine whether $|CLIQ(X_{ij})| = 2$ in constant time. Hence, it takes $|TLCL_j|$ steps to compute RS_TLCL_j from $TLCL_j$. For example, in Fig. 5, $RS\_TLCL_8 = (7,5,4,1)$ and $RS\_TLCL_4 = (3)$. For each line j, let j_p be the largest line less than j such that $i\_base[j_p] = i\_base[j]$, if any exists. Note that j_p is the largest line for which $mc2(X_{ij_p})$ is available as a result of the j_pth stage. We can use $mc2(X_{ij_p})$ to compute $mc2(X_{ij})$ without revisiting the lines less than j_p. To compute all j_p's, scan $i\_base[j]$ for j from 1 to n, and append j to $list[i]$, which is empty initially, if $i\_base[j] = i$. For each i, let $list[i] = \{t_1, \ldots, t_k\}$. Then $j_p = t_{i-1}$ if $j = t_i$, $1 < i \leq k$, and let $j_p = -1$ if $j = t_1$. This procedure takes $O(n)$ time. Note that $|CLIQ(X_{ij_p})| = 2$, because, otherwise, we would have either $i\_base[j_p] \neq i\_base[j]$ or $|CLIQ(X_{ij})| \neq 2$. For each X_{ij}, where $i \in RS\_TLCL_j$, consider the following two cases for computing $mw(mc2(X_{ij}))$, hence $mc2(X_{ij})$.

Case 1. i is the first member of RS_TLCL_j. As usual, let $X_{i_1 j_1}$ be the base of X_{ij}. Then simply visit each member of $TRCL_{i_1}$ backward starting at j until we meet j_1 or j_p. Let $(j_t = j, j_{t-1}, \ldots, j_r)$ be the list of lines visited in this order. Further consider the following subcases.

Case 1.1. $j_r = j_1$. Then clearly, as shown in Fig. 6,

$$mw(mc2(X_{ij})) = \min\{minv(X_{i_1 j_t}, minv(X_{i_1 j_{t-1}}), \ldots, minv(X_{i_1 j1})\}.$$

For example, in Fig. 5, $mw(mc2(X_{79})) = \min\{minv(X_{29}), minv(X_{28})\}$ and $mw(mc2(X_{23})) = minv(X_{23})$.

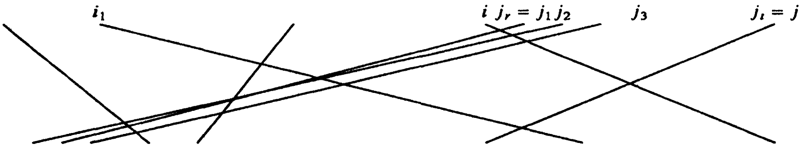


FIG. 6. *The illustration for subcase* 1.1.

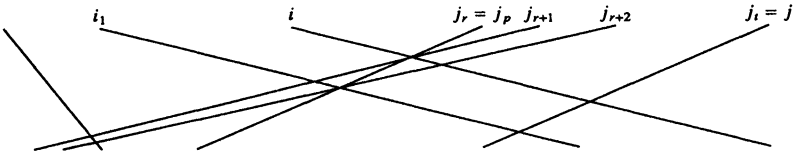


FIG. 7. *The illustration for subcase* 1.2.

Case 1.2. $j_r = j_p$. This implies that $mw(mc2(X_{ij_r}))$ is known as a result of the computation at the j_rth stage as illustrated in Fig. 7. Further, $C(X_{ij_r}) \subseteq C(X_{ij})$. Hence,

$$mw(mc2(X_{ij})) = \min\{minv(X_{i_1j_t}), minv(X_{i_1j_{t-1}}), \ldots, minv(X_{i_1j_{r+1}}), mw(mc2(X_{ij_r}))\}.$$

In Fig. 5, $mw(mc2(X_{7(10)})) = \min\{minv(X_{2(10)}), mw(mc2(X_{79}))\}$.

Case 2. i is not the first member in RS_TLCL_j. Let i' be the line that immediately precedes i in RS_TLCL_j. Then it is obvious that $C(X_{i'j}) \subseteq C(X_{ij})$ as mentioned earlier. As in Case 1, consider each member of $TRCL_{i_1}$ backward starting at j until we meet either j_1 or j_p. Again let $(j_t = j, j_{t-1}, \ldots, j_r)$ be the lines to be visited in this order. Again consider the following subcases.

Case 2.1. $j_r = j_1$. In this case, we do not actually examine all of the lines in $(j_t = j, j_{t-1}, \ldots, j_r)$ since some lines, say $(j_t = j, j_{t-1}, \ldots, j_s)$, $t \geq s \geq r$, were checked already when $mc2(X_{i'j})$ was computed at the i'th step of the jth stage. (Note that there exists at least one such line since j is one of the lines that was already computed.) In Fig. 8, only j_2 and j_1 will be examined because $j_t, j_{t-1}, \ldots, j_3$ were visited to obtain $mc2(X_{i'j})$. Hence, we compare $mw(mc2(X_{i'j}))$, $minv(X_{i_1j_{s-1}})$, $minv(X_{i_1j_{s-2}}), \ldots,$ and $minv(X_{i_1j_1})$ to pick a pair with the smallest value. In Fig. 5, $mw(mc2(X_{59})) = \min\{mw(mc2(X_{79})), minv(X_{26})\}$.

Case 2.2. $j_r = j_p$ and $i' > j_p$. Similar to subcase 2.1, let $(j_t = j, j_{t-1}, \ldots, j_s)$, $t \geq s \geq r$, be the lines already considered when $mc2(X_{i'j})$ was computed. Moreover, the lines $(j_p = j_r, j_{r-1}, \ldots, j_1)$ were already checked when $mc2(X_{ij_p})$ was computed at the ith step of the j_pth stage as shown in Fig. 9. So, simply compare $mw(mc2(X_{i'j}))$, $minv(X_{i_1j_{s-1}})$, $minv(X_{i_1j_{s-2}}), \ldots, minv(X_{i_1j_{r+1}})$, and $mw(mc2(X_{ij_r}))$ to pick a pair with the smallest value. Consider X_{58} in Fig. 5 as an example and verify that $mw(mc2(X_{58})) = \min\{mw(mc2(X_{56})), mw(mc2(X_{78}))\}$.

Case 2.3. $j_r = j_p$ and $i' < j_p$. It is not difficult to show that $mc2(X_{ij})$ is either $mc2(X_{i'j})$ or $mc2(X_{ij_r})$, whichever has a smaller mw value, since $C(X_{ij}) = C(X_{i'j}) + C(X_{ij_r})$. X_{48} in Fig. 5 is such an example. Recall that i_1 is the i-base line for both j and j_p. (See Fig. 10.)

This procedure is formalized in Algorithm 5.1.

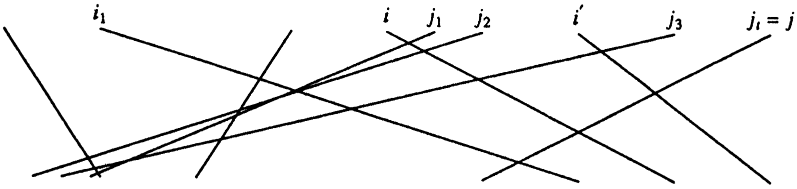


FIG. 8. *The illustration for subcase* 2.1.

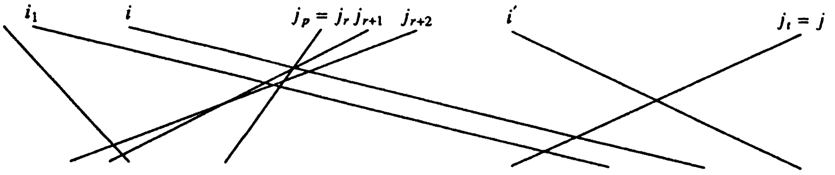


FIG. 9. *The illustration for subcase* 2.2.

ALGORITHM 5.1.

```
procedure Find mc2(j); {Compute all $mc2(X_{ij})$'s for a given $j$.}
    Compute $minv(X_{ij})$ for each $i \in BRCL_j$;
    $mw(mc2(X_{jj})) = X_{jj}$;

    {Compute all $mc2(X_{ij})$'s for $|CLIQ(X_{ij})| = 2$.}
    Let $i_1 = i\_base[j]$;
    Let $j_p < j$ be the largest line whose $i$-base is $i_1$, if exists;
        otherwise, let $j_p = -1$.
    Let $RS\_TLCL_j = (i'_m, i'_{m-1}, \ldots, i'_1)$;
    for $l = m$ downto 1 do
        Let $i = i'_l$;
        Let $j_1 = j\_base[i]$;
        Let $TRCL_{i_1}[j_1, j] = (j_1, j_2, \ldots, j_t = j)$;
        {Case 1.1} if $i = i'_m$ and $i'_m \geq j_p$ then
            $mw(mc2(X_{ij})) = \min\{minv(X_{i_1 j_t}, minv(X_{i_1 j_{t-1}}), \ldots, minv(X_{i_1 j_1})\}$;
        {Case 1.2} if $i = i'_m$ and $i'_m < j_p$ then
            $mw(mc2(X_{ij}))$
            $= \min\{minv(X_{i_1 j_t}), minv(X_{i_1 j_{t-1}}, \ldots, minv(X_{i_1 j_{p+1}}), mw(mc2(X_{ij_p}))\}$;

        Let $i' = i'_{l+1}$;
        Let $j_s$ be the smallest line in $TRCL_{i_1}[j_1, j]$ visited in the immediately pre-
            ceding step of the current stage where $mc2(X_{i'j})$ was computed;
        {Case 2.1} if $i < i'_m$ and $j_p \leq i$ then
            $mw(mc2(X_{ij}))$
            $= \min\{mw(mc2(X_{i'j})), minv(X_{i_1 j_{s-1}}), \ldots, minv(X_{i_1 j_1})\}$;
        {Case 2.2} if $i < i'_m$ and $i < j_p < i'$ then
            $mw(mc2(X_{ij})) =$
            $\min\{mw(mc2(X_{i'j})), minv(X_{i_1 j_{s-1}}), minv(X_{i_1 j_{s-2}}), \ldots, minv(X_{i_1 j_{p+1}}),$
            $mc2(X_{ij_p})\}$;
        {Case 2.3} if $i < i'_m$ and $i' < j_p$ then
```

```
            $mw(mc2(X_{ij})) = \min\{mw(mc2(X_{i'j})), mw(mc2(X_{ij_p}))\}$;

        $mw(mc2(X_{ij})) = \min\{mw(mc2(X_{ij})), mw(X_{ii}), mw(X_{jj})\}$
    endfor; $\{l\}$
```

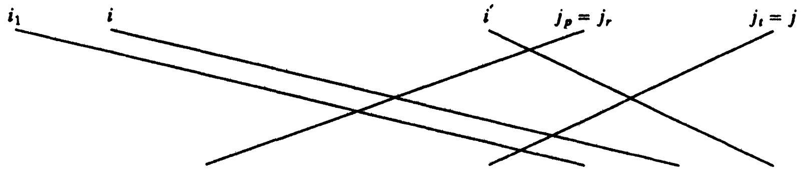


FIG. 10. *The illustration for subcase 2.3.*

TABLE 1
Computing $mw(mc2(X_{ij}))$.

j	i	$C(X_{ij})$	$mw(mc2(X_{ij}))$	Case
3	2	{23}	$minv(X_{23})$	1.1
4	3	{24, 34}	$minv(X_{24})$	1.1
6	5	{56, 26, 36, 46, 16}	$minv(X_{16})$	1.1
	4	{56, 26, 36, 46, 16}	$mw(mc2(X_{56}))$	2.1
	1	{56, 26, 36, 46, 16}	$mw(mc2(X_{46}))$	2.1
8	7	{78, 58, 28, 38, 48, 18}	$minv(X_{18})$	1.1
	5	$C(X_{56}) + C(X_{78})$	$\min\{mw(mc2(X_{56})), mw(mc2(X_{78}))\}$	2.2
	4	$C(X_{46}) + C(X_{58})$	$\min\{mw(mc2(X_{46})), mw(mc2(X_{58}))\}$	2.3
	1	$C(X_{16}) + C(X_{48})$	$\min\{mw(mc2(X_{16})), mw(mc2(X_{48}))\}$	2.3
9	7	{79, 59, 29, 78, 58, 28}	$\min\{minv(X_{29}), minv(X_{28})\}$	1.1
	5	$C(X_{79})+$ {56, 26}	$\min\{mw(mc2(X_{79})), minv(X_{26})\}$	2.1
	2	$C(X_{59}) + \{24\} + C(X_{23})$	$\min\{mw(mc2(X_{59})), minv(X_{24}), mw(mc2(X_{23}))\}$	2.2
10	7	$\{7(10), 5(10), 2(10)\} + C(X_{79})$	$\min\{minv(X_{2(10)}), mw(mc2(X_{79}))\}$	1.2
	5	$C(X_{7(10)}) + C(X_{59})$	$\min\{mw(mc2(X_{7(10)})), mw(mc2(X_{59}))\}$	2.3
	2	$C(X_{5(10)}) + C(X_{29})$	$\min\{mw(mc2(X_{5(10)})), mw(mc2(X_{29}))\}$	2.3

Table 1 shows the computational relationship among all $mw(C(X_{ij}))$'s of our sample graph in Fig. 5 for $|CLIQ(X_{ij})| = 2$.

THEOREM 5.1. *All $mc2(X_{ij})$'s can be computed in $O(m + n)$ time.*

Proof. Since the correctness of our algorithm follows from the preceding discussion, we show its time complexity. As shown previously, all RS_TLCL_j's and j_p's can be obtained in $O(m + n)$ time and $O(n)$ time, respectively. All $minv(X_{ij})$'s can be obtained in $O(n+m)$ time using the prefix computation. It then suffices to show that each $minv(X_{ij})$ value is accessed at most once in calculating all $mc2(X_{ij})$'s in our algorithm that computes all $back(X_{ij})$'s. Let $minv(X_{i_1j'})$ be an arbitrary $minv$ value accessed in computing for some X_{ij} if any exists. Then $i\_base[j] = i_1$. We show that such X_{ij} is unique.

Let $K = (k_1, \ldots, k_s)$ be the sublist of $TRCL_{i_1}$ such that $i_1 < k_1 < \cdots < k_s$ and $|CLIQ(X_{i_1k_r})| = 2$ for $1 \leq r \leq s$. Let $H = (h_1, \ldots, h_t)$ be the sublist of $TLCL_{k_s}$ such that $i_1 = h_1 < \cdots < h_t < k_s$ and $|CLIQ(X_{h_rk_s})| = 2$ for each $r = 2, 3, \ldots, t$. Then it is readily seen that K and H are disjoint and no two lines in K (and H) intersect each other, as shown in Fig. 11. Let $L = (l_1, \ldots, l_{s+t})$ be the list obtained by merging

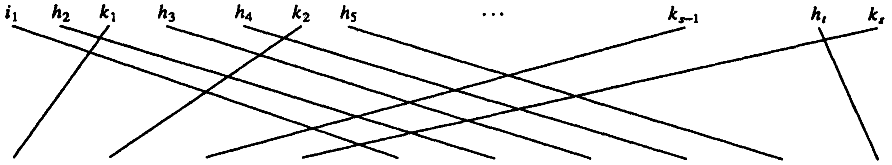


FIG. 11. *An illustration for K and H.*

K and H in increasing order. Now, try to position j' in L. If $j' = l_1 (= i_1)$, then $X_{i_1 j'}$ is used only in computing $mc2(X_{i_1 i_1})$. So, we assume that $j' > l_1$. Let l_r be the largest line smaller than j' in L. Then, we consider the following cases.

(1) $l_r = h_u$ and $l_{r+1} = k_v$ for some $1 \le u \le t$ and $1 \le v \le s$. Then $minv(X_{i_1 j'})$ is accessed only in computing $mc2(X_{h_u k_v})$. Hence X_{ij} is $X_{h_u k_v}$.

(2) $l_r = h_u$ and $l_{r+1} = h_{u+1}$ for some $1 \le u < t$. In this case, let k' be the smallest line in K such that $k' > h_{u+1}$. Then $minv(X_{i_1 j'})$ is accessed only in computing $mc2(X_{h_u k'})$. Hence X_{ij} is $X_{h_u k'}$.

(3) $l_r = k_v$ and $l_{r+1} = h_u$ for some $1 \le v \le s$ and $1 \le u \le t$. In this case, let k' be the smallest line in K such that $k' > h_u$, and let h' be the largest line in H such that $h' < k_v$. Then X_{ij} is $X_{h'k'}$.

(4) $l_r = k_v$ and $l_{r+1} = k_{v+1}$ for some $1 \le v < s$. Let h' be the largest line in H such that $h' < k_v$. Then X_{ij} is $X_{h' k_{v+1}}$.

(5) $l_r = l_{s+t}$. Then X_{ij} is $X_{l_{s+t} j'}$.

Clearly such X_{ij} is unique. Thus the number of $minv(X_{ij})$'s used in computing for all $mc2(X_{ij})$'s is $O(m+n)$. Since at most two previous computed $mc2(X_{ij})$ values are involved in computing a current $mc2(X_{ij})$ value, the total number of $mc2(X_{ij})$'s used in computing for all $mc2(X_{ij})$'s is limited to $O(m+n)$. Therefore, it takes $O(m+n)$ time to compute all $mc2(X_{ij})$'s. □

6. Summary.

We first review the dynamic strategy for computing $back(X_{ij})$ in Algorithm 3.1. The ordered lists ($TLCL_j$, $TRCL_j$, $BLCL_j$, and $BRCL_j$), $limit(X_{ij})$, and $cdiv(X_{ij})$ are computed during the initialization process. Their computational procedures are described in Lemmas 4.1, 4.2, and 4.3, respectively. At the jth stage, $j = 1, 2, \ldots, n+1$, our algorithm computes $mw(X_{ij})$ and $back(X_{ij})$ for each $i \in TLCL_j$ in increasing order. Each $mc(limit(X_{ij}))$ is available, and both $iback(X_{ij})$ and $jback(X_{ij})$ can be computed as described in Lemma 4.4. These functions are used to compute $back(X_{ij})$. Next in the same stage j, we compute $minv(X_{ij})$'s, $mc2(X_{ij})$'s, and $mc(X_{ij})$'s. All $minv(X_{ij})$'s are computed in reverse order of the list $BRCL_j$, and $mc2(X_{ij})$'s are computed for all i in reverse order of the list $TLCL_j$. The algorithm for computing $mc2(X_{ij})$ is described in Algorithm 5.1. Finally, $mc(X_{ij})$ are computed for all i in reverse order of the list $TLCL_j$. If $|CLIQ(X_{ij})| \le 2$, then $mc(X_{ij})$ is equivalent to one of $mc2(X_{ij})$, X_{ii} and X_{jj}, for which the mw value is a minimum. On the other hand, if $|CLIQ(X_{ij})| > 2$, then $mc(X_{ij})$ is either $mc(X_{ik})$ or $mc(X_{kj})$, whichever has a smaller mw value, where $k = cdiv(X_{ij})$. Now we are ready to state the following lemmas.

LEMMA 6.1. *All $mc(X_{ij})$'s can be found in $O(m+n)$ time.*

Proof. According to Theorem 3.1, if $cdiv(X_{ij}) = k (\ne i)$, then $mc(X_{ij}) = X_{i'j'} \in \{mc(X_{ik}), mc(X_{kj})\}$ such that $mw(X_{i'j'})$ is the smaller of $mw(mc(X_{ik}))$ and $mw(mc(X_{kj}))$. Otherwise, i.e., $cdiv(X_{ij}) = i$, $mc(X_{ij}) = mc2(X_{ij})$. Since all $cdiv(X_{ij})$'s can be found in $O(m+n)$ time (Lemma 4.3) and all $mc2(X_{ij})$'s can also be computed in $O(m+n)$ time (Theorem 5.1), it follows that all $mc(X_{ij})$'s can be found in $O(m+n)$ time. □

LEMMA 6.2. *All $back(X_{ij})$'s can be obtained in $O(m+n)$ time.*

Proof. For each X_{ij} we first identify $mc(limit(X_{ij}))$, which was computed at a previous stage. Then $back(X_{ij})$ is $iback(X_{ij})$, $jback(X_{ij})$, or $mc(limit(X_{ij}))$, whichever has the smallest value in $W(MWDS(iback(X_{ij}))+S_{ij})$, $W(MWDS(jback(X_{ij}))+S_{ij})$, or $W(MWDS(mc(limit(X_{ij}))) + S_{ij})$. The validity of this procedure is supported by Theorems 2.2 and 3.1. □

Note that once all $back(X_{ij})$'s are computed, a minimum-weight dominating set can be obtained by tracing back through the function *back*, which takes no more than $O(n)$ time. Thus we have the following main theorem.

THEOREM 6.3. *The minimum-weight dominating set problem for permutation graphs can be solved in $O(m+n)$ time.*

7. Conclusion. In this paper, we presented an algorithm for finding a minimum-weight dominating set in a permutation graph. The algorithm takes only $O(n+m)$ time, where n is the number of nodes and m is the number of edges in the graph. The algorithm basically uses the same approach as introduced in [6]. The improvement in time complexity is achieved as a result of identifying and removing some of the redundant computations. This is the best available algorithm to date for this problem.

Acknowledgments. The authors thank the anonymous referees for their comments that greatly helped improve the presentation of this paper. We also thank Glenn Manacher for bringing [7] to our attention.

REFERENCES

[1] M. J. ATALLAH AND S. R. KOSARAJU, *An efficient algorithm for maxdominance, with applications,* Algorithmica, 4 (1989), pp. 221–236.

[2] M. J. ATALLAH, G. K. MANACHER, AND J. URRUTIA, *Finding a minimum independent dominating set in a permutation graph,* Discrete Appl. Math., 21 (1988), pp. 177–183.

[3] D. G. CORNEIL, Y. PERL, AND L. K. STEWART, *A linear recognition algorithm for cographs,* SIAM J. Computing, 14 (1985), pp. 926–934.

[4] M. FARBER AND J. M. KEIL, *Domination in permutation graphs,* J. Algorithms, 6 (1985), pp. 309–321.

[5] M. C. GOLUMBIC, *Algorithmic Graph Theory and Perfect Graphs,* Academic Press, New York, 1980.

[6] Y. D. LIANG, C. RHEE, S. K. DHALL, AND S. LAKSHMIVARAHAN, *A new approach for the domination problem on permutation graphs,* Inform. Process. Lett., 37 (1991), pp. 219–224.

[7] G. K. MANACHER AND T. A. MANKUS, *Incorporating negative-weight vertices in certain vertex-search graph algorithms,* Inform. Process. Lett., 42 (1992), pp. 293–294.

[8] A. PNUELI, A. LEMPEL, AND S. EVEN, *Transitive orientation of graphs and identification of permutation graphs,* Canad. J. Math., 23 (1971), pp. 160–175.

[9] C. RHEE, S. K. DHALL, and S. LAKSHMIVARAHAN, An *NC* algorithm for the domination in permutation graphs, in Proc. 28th Annual Allerton Conference on Communication, Control, and Computing, Monticello, IL, 1990, pp. 294–295.

[10] J. SPINRAD, *On comparability and permutation graphs,* SIAM J. Comput., 14 (1985), pp. 658–670.

[11] J. SPINRAD, A. BRANDSTÄDT, AND L. STEWART, *Bipartite permutation graphs,* Discrete Appl. Math., 18 (1987), pp. 279–292.

[12] K. H. TSAI AND W. L. HSU, *Fast algorithms for the dominating set problem on permutation graphs,* in SIGAL'90 Algorithms, Asano et al., eds., Lecture Notes in Computer Science, Springer-Verlag, New York, 450 (1990), pp. 109–117.

SIAM J. COMPUT.
Vol. 25, No. 2, pp. 420–447, April 1996

012

GENERALIZED KRAFT'S INEQUALITY AND DISCRETE k-MODAL SEARCH*

ANMOL MATHUR† AND EDWARD M. REINGOLD‡

Abstract. A function $f : \Re \to \Re$ is *k-modal* if its kth derivative has a unique zero. We study the problem of finding the smallest possible interval containing the unique zero of the kth derivative of such a function, assuming that the function is evaluated only at integer points. We present optimal algorithms for the case when k is even and for $k = 3$ and near-optimal algorithms when $k \geq 5$ and odd. A novel generalization of Kraft's inequality is used to prove lower bounds on the number of function evaluations required. We show how our algorithms lead to an efficient divide-and-conquer algorithm to determine all turning points or zeros of a k-modal function. Unbounded k-modal search is introduced and some problems in extending previous approaches for unbounded searching to the k-modal case are discussed.

Key words. k-modal search, Kraft's inequality, k-modal functions, optimal algorithms, binary search, unimodal search, bimodal search, unbounded search, Fibonacci numbers

AMS subject classifications. 68Q25, 68Q20, 11B39

1. Introduction.

The problem of determining properties of an unknown function by evaluating it at scattered points arises in many different contexts. If the function is arbitrary, then this might require the evaluation of the function at all points in its domain. However, if the function is well behaved, it is possible to determine some parameters of the function from only a few evaluations. Shannon's sampling theorem in communication theory is a classic example of this problem. More recently, geometric probing has become an active area of research in computational geometry [20]; such probing aims to find some shape parameters of an object by probing it at the fewest possible points. In this paper we present a natural measure of the complexity of a function called the *modality* and demonstrate its utility in the design of efficient algorithms for finding certain parameters of the function.

A function $f : \Re \to \Re$ is *k-modal* if its kth derivative has a unique zero; such a function can have at most k local extrema. Although the use of local extrema in this definition suggests an implicit assumption about the continuity of f, the definition is easily extended to discrete functions $f : \{0, 1, \ldots, N\} \to \Re$. The analog of the kth derivative in the discrete case is $\Delta^k f$, defined recursively by

$$\Delta^k f = \Delta(\Delta^{k-1} f),$$
$$\Delta f(i) = f(i+1) - f(i).$$

With this in mind, we will use the term "kth derivative of f" to refer to the true kth derivative of f if f is sufficiently differentiable and to $\Delta^k f$ if f is discrete. A "zero" of a discrete function is an integer $i, 0 \leq i < N$ such that either

$$f(i) = 0$$

* Received by the editors March 29, 1993; accepted for publication (in revised form) September 7, 1994.

† Department of Computer Science, University of Illinois at Urbana-Champaign, 1304 West Springfield Avenue, Urbana, IL 61801. The research of this author was supported in part by National Science Foundation grant MIP-92-22408.

‡ Department of Computer Science, University of Illinois at Urbana-Champaign, 1304 West Springfield Avenue, Urbana, IL 61801. The research of this author was supported in part by National Science Foundation grant CCR-93-20577.

or

$$f(i) \cdot f(i+1) < 0.$$

We define a *turning point* of the discrete function f as an integer $i, 0 \leq i < N$, such that

$$\Delta f(i-1) \cdot \Delta f(i) < 0.$$

A discrete k-modal function can have at most k turning points (this follows from Lemma 2.2 below). It is important to note that a k-modal function can have fewer than k turning points (see Fig. 1). One major class of continuous k-modal functions is the set of polynomials of degree $k+1$. A discrete function that is obtained by restricting a continuous k-modal function to a set of points so that at least one point lies between any two adjacent roots of the continuous function is k-modal.

Throughout this paper, we assume that for a k-modal function f over the interval $[0, N)$, $\Delta^k f$ is increasing[1] and $f(0) = (-1)^{k+1}\infty$, $f(N) = (-1)^k\infty$. We define the *discrete k-modal search problem* as the determination of an interval $[i, i+k+1)$ containing the unique zero of the kth derivative of f by evaluating (probing) the function only at integer points in the range $[0, N)$. The smallest range to which the unique zero of the kth derivative of f can be localized is of size $k+1$ because at least $k+1$ values of f are required to compute a single value of $\Delta^k f$.

The problem of finding the zeros of continuous functions in one variable is closely related to its discrete counterpart in the case when only function evaluations are allowed [14], [19]. In order to approximate a zero of a continuous function $f : [0,1] \rightarrow \Re$ to an accuracy $\epsilon > 0$, we can locate the zero of the discrete function obtained by sampling the continuous function at $i\epsilon, 0 \leq i \leq 1/\epsilon$. Thus, the lower bounds and algorithms presented in this paper apply to continuous functions if the discretized function is k-modal.

The case $k = 0$ is the problem of finding the zero of a monotonic function, for which binary search is the optimal strategy. The case $k = 1$ corresponds to searching for the unique local maximum/minimum of a unimodal function. This problem was first studied by Kiefer [10], who gave an optimal algorithm for narrowing, in n probes, a unit interval containing the local extremum to an interval of length at most $1/F_{n+1} + \epsilon$, for any fixed $\epsilon > 0$, where F_i is the ith Fibonacci number defined by $F_0 = 0, F_1 = 1$, and $F_i = F_{i-1} + F_{i-2}$. Kiefer proved that this *Fibonacci search* is optimal in the sense that any algorithm using n probes that narrows the interval containing the local extremum to at most $1/F_{n+1}$ for some functions must fail to narrow the length of the interval to at most $1/F_{n+1} + \epsilon$ for other functions. Further work on variants of the unimodal search problem was done by Avriel and Wilde [2], Oliver and Wilde [15], and Witzgall [23]. Karp and Miranker [9] considered the unimodal searching problem under the assumption that several probes can be done in parallel. More recently, Goldstein and Reingold [6] studied the discrete unimodal search problem in which the probes are restricted to integer points. They use a variant of Kraft's inequality to prove a lower bound on the number of function evaluations required for the discrete unimodal search problem. We generalize their results.

Interest in discrete k-modal searching arises because of its applications in computational geometry, especially in algorithms for intersection of convex objects in two

[1] This does not entail any loss of generality because if $\Delta^k f$ is decreasing, then $\Delta^k(-f)$ is increasing. So, we can determine whether $\Delta^k f$ is increasing or decreasing by looking at the sign of $f(0)$ and, if it is decreasing, substitute $-f$ for f.

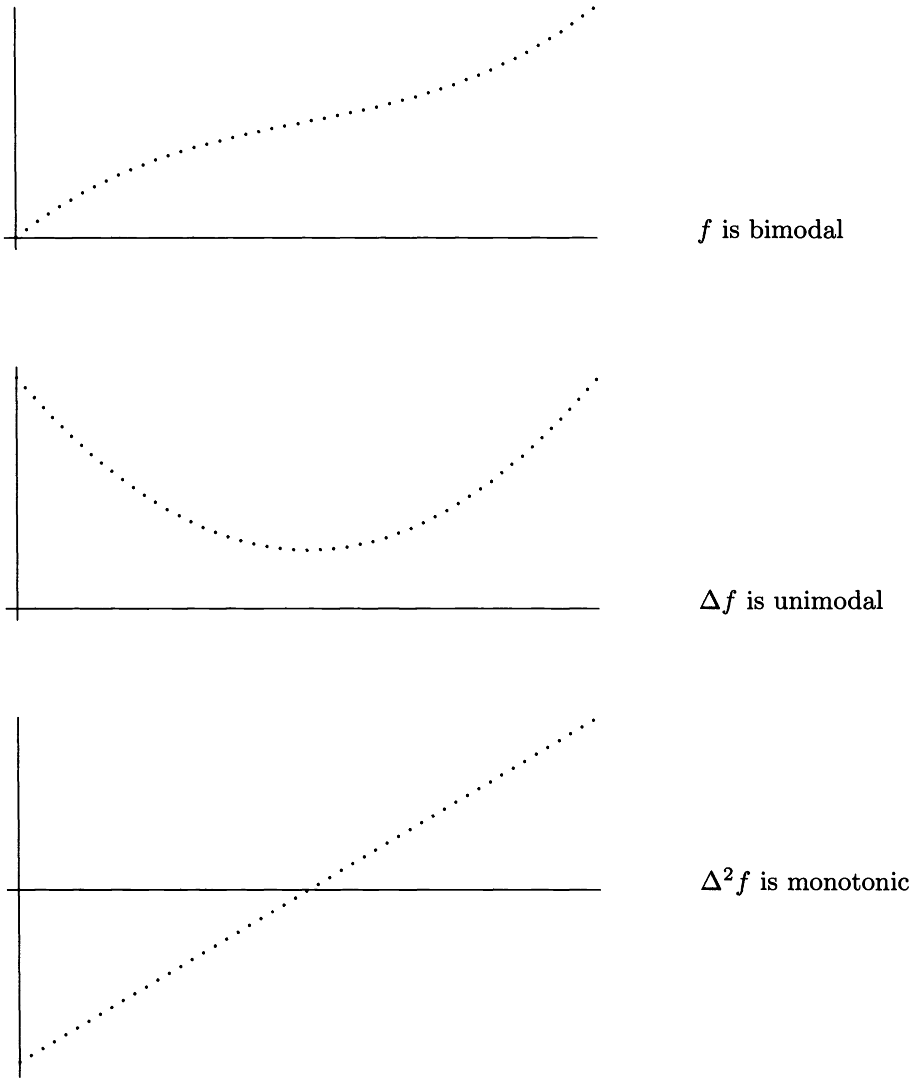


FIG. 1. *Example of a bimodal function. Notice that because $\Delta f > 0$, f has no turning points.*

and three dimensions [1], [4]. However, the case of k-modal searching for $k \geq 2$ has received little attention. Veroy [22] studied the problem of finding the extrema of a continuous, periodic bimodal function. Hyafil [7] studied the continuous k-modal search problem and proposed optimal algorithms for even k. He used the relationship between the complexity of finding the unique zero of a monotonic function with k asynchronous processors and that of $(k-1)$-modal searching with a single processor to prove lower bounds on the number of function evaluations required for k-modal search. This technique for proving lower bounds was first used by Kung [12]. In [13], Kung makes use of adversary arguments to prove lower bounds on the complexity of finding a zero of a continuous function. In contrast, we use characteristics of k-modal search trees to prove a generalization of the Kraft's inequality. The k-modal search algorithms and the algorithm for finding all the turning points of a discrete k-modal function have applications in one-dimensional optimization problems for which the objective function is not available in closed form and/or is time consuming to evaluate.

In this paper we extend the algorithms of [7] to discrete k-modal search and use a novel generalization of Kraft's inequality to give lower bounds on the number of function evaluations required for discrete k-modal search; a related, continuous generalization of Kraft's inequality was used by Karp [8] to derive lower bounds on the cost of some lopsided prefix-free codes. In the process of our work, some identities are proved for a natural generalization of the Fibonacci numbers. One consequence of our work is an improvement to an algorithm in [4] which uses bimodal searching ($k = 2$). We also present an algorithm to find all the local extrema of a k-modal function.

The organization of the rest of the paper is as follows: §2 discusses the representation of k-modal search algorithms as k-modal search trees and formulates a theorem that is used to prove lower bounds on the number of probes required for k-modal search. In §3 we prove the theorem establishing the lower bounds. Section 4 discusses algorithms for k-modal search and §5 describes how to use those algorithms to design an efficient algorithm for finding all the extrema of a k-modal function. Section 6 discusses unbounded k-modal search. Finally, in §7, we conclude with some open problems.

2. k-modal search trees and lower bounds.

All algorithms that use only function evaluations for k-modal searching maintain an *interval of uncertainty*, $I \subseteq [0, N)$, that is guaranteed to contain the unique zero of the kth derivative of the function. The algorithms begin with $I = [0, N)$ and use function evaluations to prune I until it becomes sufficiently small. These algorithms rely on the following lemmas.

LEMMA 2.1. *Given a k-modal function f, the knowledge of the function at fewer than $k + 1$ points* inside *the interval of uncertainty cannot reduce the length of the interval of uncertainty.*

Proof. It is easy to see that for a monotonic function, the interval of uncertainty cannot be shortened without making at least one probe. Since the kth derivative of a k-modal function f is monotonic and at least $k+1$ values of f are required to compute one value of the kth derivative of f, it follows that at least $k + 1$ values of f inside the interval of uncertainty are required to shorten it. □

LEMMA 2.2 (discrete Rolle's theorem). *Given a discrete function*

$$f : \{0, 1, \ldots, N\} \rightarrow \Re$$

having zeros at i and j, $i < j$, there is a k, $i \leq k \leq j$, such that Δf has a zero at k.

Proof. Assume without loss of generality that j is the first zero of f greater than i (otherwise one can replace j with the first zero greater than i in the subsequent argument). Since i is a zero of f, either $f(i) = 0$ or $f(i) \cdot f(i + 1) < 0$. Without loss of generality, assume that $f(i + 1) \geq 0$. If $f(i) = f(i + 1) = 0$, then $\Delta f(i) = 0$ and we are done. Otherwise, $f(i) < 0$ and $f(i + 1) > 0$, and then $\Delta f(i) > 0$. Since there is no other zero of f between i and j, $f(j) \geq 0$ and $f(j + 1) \leq 0$. Again, if $f(j) = f(j + 1) = 0$, then j is a zero of Δf and we are done; otherwise $\Delta f(j) < 0$. Hence, there must be some k, $i \leq k < j$ such that either $\Delta f(k) = 0$ or $\Delta f(k) > 0$ and $\Delta f(k + 1) < 0$. But such a k is a zero of Δf, thus proving the theorem. □

LEMMA 2.3. *Let $f : \{0, 1, \ldots, N\} \rightarrow \Re$ be a discrete function such that $\Delta^k f$ is increasing and $\Delta^k f(i) > 0$, for all $i \in \{0, 1, \ldots, N - k\}$. Then for every set $\{x_0, x_1, \ldots, x_k\}$ of $k + 1$ distinct values in $\{0, \ldots, N\}$,*

$$P(x_0, x_1, \ldots, x_k, f) > 0,$$

where

$$P(x_0, x_1, \ldots, x_k, f) = \sum_{i=0}^{k} \frac{f(x_i)}{\prod_{j \neq i}(x_j - x_i)}.$$

Proof. Consider the Lagrange interpolation formula [5] for the set of points $(x_i, f(x_i)), 0 \leq i \leq k$, given by $\sum_{i=0}^{k} f(x_i) \prod_{j \neq i} \frac{x - x_j}{x_i - x_j}$ and its error

$$E(x) = f(x) - \sum_{i=0}^{k} f(x_i) \prod_{j \neq i} \frac{x - x_j}{x_i - x_j}.$$

Since $E(x_i) = 0$, $0 \leq i \leq k$, by repeated application of Lemma 2.2, we can conclude that $\Delta^k E$ has a zero at some $m \in \{0, \ldots, N-k\}$. Furthermore, since Δ distributes over summation and $\Delta^k x^l = 0$, if $l < k$, we have

$$(1) \qquad \Delta^k f(x) = \Delta^k E(x) + k! P(x_0, x_1, \ldots, x_k, f).$$

By assumption, $\Delta^k f$ is positive at all points in $\{0, \ldots, N-k\}$. Also, since m is a zero of $\Delta^k E$, either $\Delta^k E(m) = 0$ or one of $\Delta^k E(m)$ or $\Delta^k E(m+1)$ is negative. Hence, from equation (1) we get that $P(x_0, x_1, \ldots, x_k, f) > 0$. □

LEMMA 2.4. *Given a k-modal function f and $k+3$ of its values*

$$f(x_0), f(x_1), \ldots, f(x_{k+2}),$$

with

$$x_0 < x_1 < \cdots < x_{k+1} < x_{k+2},$$

the interval of uncertainty can always be reduced to $[x_0, x_{k+1})$ or $[x_1, x_{k+2})$ irrespective of the choice of $x_1, \ldots, x_{k+2}$. Furthermore, in the worst case, the interval of uncertainty cannot be shortened by more than this amount.

Proof. Suppose the unique zero of $\Delta^k f$ lies in $[x_0, x_1)$. Then, $\Delta^k f$ is positive in $[x_1, N)$. So, using Lemma 2.3,

$$P(x_1, x_2, \ldots, x_{k+1}, f) > 0.$$

Thus, if

$$P(x_1, x_2, \ldots, x_{k+1}, f) \leq 0,$$

we can conclude that the unique zero of $\Delta^k f$ does not lie in $[x_0, x_1)$. By a symmetric argument, if $P(x_1, x_2, \ldots, x_{k+1}, f) \geq 0$, then the unique zero of $\Delta^k f$ does not lie in $[x_{k+1}, x_{k+2})$; see Fig. 2. If $P(x_1, x_2, \ldots, x_{k+1}, f) = 0$, then both $[x_0, x_1)$ and $[x_{k+1}, x_{k+2})$ can be pruned.

Given $k+3$ values of f such that $P(x_1, x_2, \ldots, x_{k+1}, f) \leq 0$, one can construct a function g which has the same values as f at the $k+3$ specified points but has the unique zero of its kth derivative anywhere in $[x_1, x_{k+2})$. Thus, no algorithm for discrete k-modal search can shorten the interval of uncertainty to be smaller than $[x_1, x_{k+1})$, given only the values of the function at $x_i, 0 \leq i \leq k+2$. A symmetric argument works for the case when the P function is nonnegative. □

Thus, the algorithms for k-modal search maintain an interval of uncertainty $I = [x_0, x_{k+1})$ in the form of an array of positions $[x_0, x_1, \ldots, x_k, x_{k+1}]$ at which

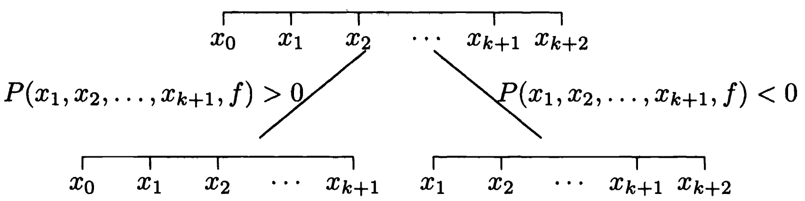

x_0 x_1 x_2 $\cdots$ x_{k+1} x_{k+2}
$P(x_1, x_2, \ldots, x_{k+1}, f) > 0$
$P(x_1, x_2, \ldots, x_{k+1}, f) < 0$
x_0 x_1 x_2 $\cdots$ x_{k+1}
x_1 x_2 $\cdots$ x_{k+1} x_{k+2}

FIG. 2. *Illustration of Lemma 4; if $P(x_1, x_2, \ldots, x_{k+1}, f) = 0$, then both the leftmost and rightmost segments can be discarded.*

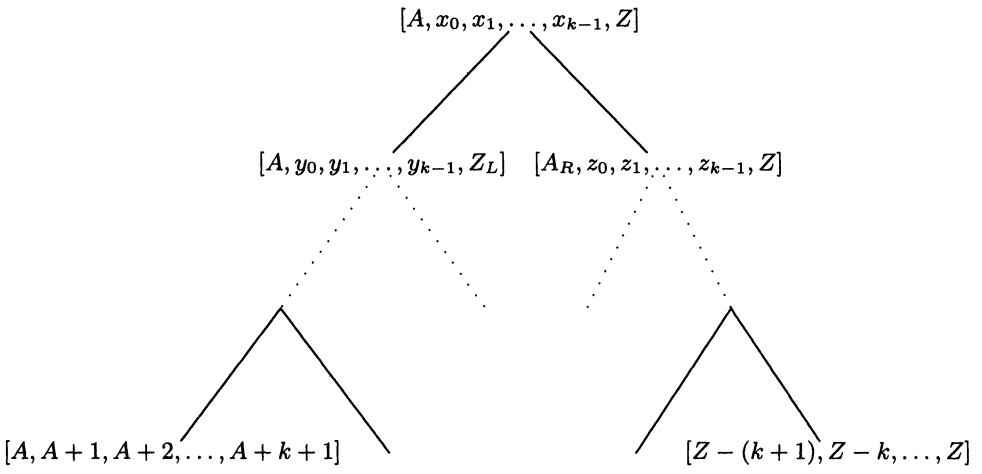

$[A, x_0, x_1, \ldots, x_{k-1}, Z]$
$[A, y_0, y_1, \ldots, y_{k-1}, Z_L]$
$[A_R, z_0, z_1, \ldots, z_{k-1}, Z]$
$[A, A+1, A+2, \ldots, A+k+1]$
$[Z-(k+1), Z-k, \ldots, Z]$

FIG. 3. *A subtree of a k-modal search tree. The path from the root to the leftmost (rightmost) leaf is the left (right) spine.*

the function has been evaluated; at each step another probe is made inside I and either the leftmost or the rightmost segment is discarded, depending on the particular values of f at the $k+3$ points in the interval at which its values are known.

Lemmas 2.1 and 2.4 tell us that an algorithm that solves the discrete k-modal search problem by probing can be described by a k-modal search tree as shown in Fig. 3. Each node is an array of the form $[A, x_0, x_1, \ldots, x_{k-1}, Z]$ and represents the current interval of uncertainty $[A, Z)$ and the distribution of probes within it, with $A = x_{-1}$ and $Z = x_k$. The left (right) child of a node represents the new interval of uncertainty after the rightmost (leftmost) segment is discarded following a new probe. If the new probe is made at y, $x_i < y < x_{i+1}$, $-1 \leq i \leq k-1$, then the left child is $[A, y_0, y_1, \ldots, y_{k-1}, Z_L]$ with $A = y_{-1}$ and $Z_L = y_k$, where $y_j = x_j, -1 \leq j \leq i$, $y_{i+1} = y$, and $y_j = x_{j-1}, i+2 \leq j \leq k$. Similarly, if $[A_R, z_0, z_1, \ldots, z_{k-1}, Z]$ is the right child with $A_R = z_{-1}$ and $Z = z_k$, then $z_j = x_{j+1}, -1 \leq j \leq i-1$, $z_i = y$, and $z_j = x_j, i+1 \leq j \leq k$. The root of the tree is $[0, w_0, \ldots, w_{k-1}, N]$, where $w_0, w_1, \ldots, w_{k-1}$ are the initial k probes, and the leaves all have the form $[l, l+1, l+2, \ldots, l+k+1]$. The leftmost (rightmost) path in a k-modal search tree is referred to as the *left* (*right*) *spine*.

The *cost* of i in a given k-modal search tree, $c(i)$, is k plus the longest distance from the root to a leaf node containing the interval $[i, i+1)$. The k initial probes contribute the additive factor of k to the cost. $c(i)$ is the maximum number of probes required for k-modal search using the k-modal search algorithm corresponding to the given k-modal search tree, if the unique zero of the kth derivative of f is in the interval $[i, i+1]$.

We are now ready to state the main theorem that establishes the lower bounds; it is a generalization, to k-modal search trees, of Kraft's inequality [11] and the Fibonacci–Kraft inequality of [6].

THEOREM 2.5. *In a k-modal search tree over the range $[0, N)$,*

$$\sum_{i=0}^{N-1} \frac{1}{L_{c(i)+K}} \leq 1, \tag{2}$$

where

$$L_n = L_{n-1-\lfloor k/2 \rfloor} + L_{n-1-\lceil k/2 \rceil}, \tag{3}$$

and if k is even, then $L_i = 1$ for $0 \leq i \leq \lceil k/2 \rceil$, while if k is odd, then $L_0 = 0$ and $L_i = 1$ for $1 \leq i \leq \lceil k/2 \rceil$. In (2),

$$K = L^{-1}(k) + 1. \tag{4}$$

So, K is the smallest integer for which $L_{K-1} \geq k$.

COROLLARY 2.6 (see [11]). *For arbitrary binary trees* (*which correspond to* 0-*modal search trees*),

$$\sum_{i=0}^{N-1} \frac{1}{2^{c(i)}} \leq 1.$$

COROLLARY 2.7 (see [6]). *For a unimodal search tree over the range $[0, N)$,*

$$\sum_{i=0}^{N-1} \frac{1}{F_{c(i)+2}} \leq 1,$$

where F_i is the ith Fibonacci number.

The lower bound on the number of probes required is given by the following corollary.

COROLLARY 2.8. *In k-modal searching, in the worst case the minimum number of probes required to narrow the interval of uncertainty from $[0, N)$ to an interval of length $k+1$ is at least $L^{-1}(N) - K$. Thus, if $k = 2p$ is even, the minimum number of probes is at least $(p+1)\lg N - K$, and if $k = 2p+1$ is odd, then the minimum number of probes is at least $(\log_\gamma 2)\lg N - K$, where γ is the dominant root of $x^{p+2} - x - 1 = 0$.*

Proof. Let $h = \max_{0 \leq i < N} c(i)$. Theorem 2.5 implies that $N/L_{h+K} \leq 1$. Hence, $h \geq L^{-1}(N) - K$. The actual values of $L^{-1}(N)$ when k is even or odd follow from the definition of the sequence L. □

COROLLARY 2.9. *In binary* (0-*modal*) *search, the minimum number of probes required to narrow the interval of uncertainty from $[0, N)$ to an interval of length* 1 *is at least $\lceil \lg N \rceil$.*

COROLLARY 2.10 (see [6]). *In unimodal search, the minimum number of probes required to narrow the interval of uncertainty from $[0, N)$ to an interval of length* 2 *is at least $F^{-1}(N) - 2$, where $F^{-1}(N)$ is the smallest i such that $F_i \geq N$.*

COROLLARY 2.11. *In bimodal search, the minimum number of probes required to narrow the interval of uncertainty from $[0, N)$ to an interval of length* 3 *is at least* $2\lceil \lg N \rceil - 3$.

3. Proof of Theorem 2.5. In this section we prove Theorem 2.5, establishing the lower bound on the number of probes for k-modal search. The proof of the theorem is by induction on the height of the k-modal search tree. The proof for the case when k is even is much simpler since the recurrence relation in L_n is simple for this case. For odd k we must prove a stronger version of the generalized Kraft's inequality in Theorem 2.5, and the proof is more complicated. In both the cases, the proof makes use of the observation that the intervals of uncertainty at nodes that are not too far down the left and right spines overlap. This allows the sum in the generalized Kraft's inequality to be split into two sums that can be bounded by induction. The following lemma formalizes the crucial observation regarding overlap of intervals of uncertainty.

LEMMA 3.1. *Let $[A, Z)$ be the interval of uncertainty at the root of a k-modal search tree and let $[A, Z_L)$ and $[A_R, Z)$ be the intervals of uncertainty at the nodes at distance d_L and d_R along the left and right spines, respectively. If*

$$d_L + d_R \leq k + 2,$$

then

$$[A, Z_L) \cup [A_R, Z) = [A, Z)$$

and

$$Z_L - A_R \geq k + 2 - (d_L + d_R).$$

Proof. Let $x_0, x_1, \ldots, x_k$ be the $k + 1$ probes *inside* the interval of uncertainty $[A, Z)$ at the root of the k-modal search tree; these $k+1$ probes include the additional probe made just before one of the extreme segments is discarded. By Lemma 2.1, in one step down any path in a k-modal search tree at most one of the extreme segments can be discarded. Hence, after d_L steps down the left spine, at most d_L of the rightmost segments, out of the $k + 2$ initial segments, can be discarded. So, $Z_L \geq x_{k-d_L+1}$. Similarly, the left endpoint of the interval of uncertainty of the node d_R steps down the right spine must be at or to the left of x_{d_R-1}. If

$$d_L + d_R \leq k + 2,$$

then

$$d_R - 1 \leq k - d_L + 1.$$

Hence,

$$A_R \leq x_{d_R-1} \leq x_{k-d_L+1} \leq Z_L.$$

So,

$$[A, Z_L) \cup [A_R, Z) = [A, Z).$$

Furthermore,

$$\begin{aligned} Z_L - A_R &\geq x_{k-d_L+1} - x_{d_R-1} \\ &\geq k - d_L + 1 - (d_R - 1) \\ &= k + 2 - (d_L + d_R), \end{aligned}$$

as claimed. □

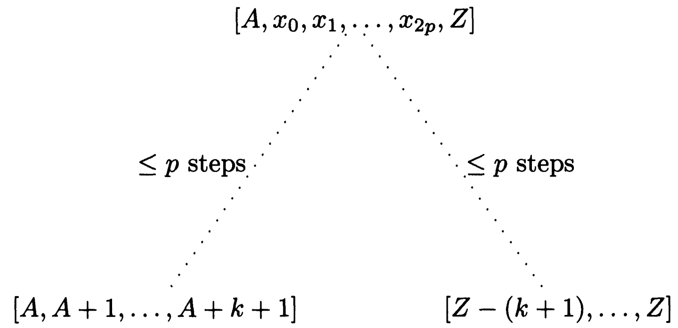


FIG. 4. *Base case for the inductive proof for even k.*

3.1. k even. We first note that for the case when $k = 2p$, the recurrence in equation (3) reduces to

$$L_n = 2L_{n-(p+1)},$$

with $L_i = 1$, $0 \leq i \leq p$, as the initial conditions. The solution to this recurrence is clearly

$$L_n = 2^{\lfloor \frac{n}{p+1} \rfloor}.$$

For the base case of the induction, the subtree has both the left and right spines of length at most p (see Fig. 4). Consider the extreme case when both the left and the right spines have length p. Let $[A, Z)$ be the interval of uncertainty at the root and let $x_0, x_1, \ldots, x_{2p}$ be the $2p+1$ probes *inside* the interval of uncertainty, including the probe made just before the first outermost interval is discarded. By Lemma 3.1, if $[A, Z_L)$ and $[A_R, Z)$ are the intervals of uncertainty at the leaves at the end of the left and right spines, respectively, then these two intervals overlap and $A_R - Z_L \geq 2p + 2 - 2p = 2$. Since both the intervals $[A, Z_L)$ and $[A_R, Z)$ correspond to leaves of the k-modal search tree, each has length $k+1$. So, the length of the interval $[A, Z)$ is at most $2(k+1) - 2 = 2k$. Also,

$$L_{k+L^{-1}(k)+1} \geq 2k.$$

Hence,

$$\sum_{i=A}^{Z-1} \frac{1}{L_{c(i)+K}} \leq \frac{2k}{L_{k+L^{-1}(k)+1}} \leq 1.$$

The theorem is thus true for all the base cases.

We now assume that the theorem is true for subtrees with height at most h, $h \geq p+1$. There are two cases for the induction step.

3.1.1. Case 1. Suppose that both left and right spines have length at least $p+1$. Here the situation is as shown in Fig. 5. Let $[A, Z)$ be the initial interval of uncertainty and let $x_0, x_1, \ldots, x_{2p}$ be the initial probes inside $[A, Z)$, including the probe made just before the first step down the tree. If $[A, Z_L)$ and $[A_R, Z)$ are the intervals of uncertainty after $p+1$ steps down the left and right spines, respectively, then by Lemma 3.1, $[A, Z_L)$ and $[A_R, Z)$ overlap. Hence, the subtrees T_L and T_R cover the

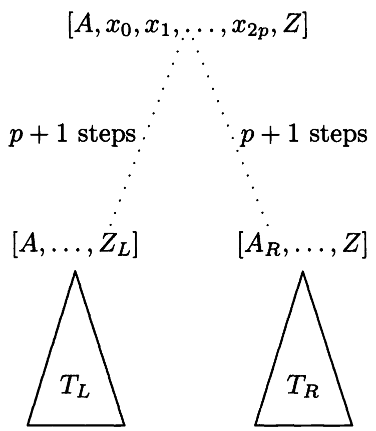


FIG. 5. *Case* 1 *of the induction for even* k.

entire range $[A, Z)$. Let c_L and c_R be the cost functions in T_L and T_R, respectively; then, using the induction hypothesis on T_L and T_R, we have

$$\text{sum in } T_L = \sum_{i=A}^{Z_L-1} \frac{1}{L_{c_L(i)+K}} \le 1$$

and

$$\text{sum in } T_R = \sum_{i=A_R}^{Z-1} \frac{1}{L_{c_R(i)+K}} \le 1.$$

Since $c(i) \ge \max\{c_L(i), c_R(i)\} + (p+1)$ and $L_{j+(p+1)} = 2L_j$, we have

$$\sum_{i=A}^{Z-1} \frac{1}{L_{c(i)+K}} \le \frac{1}{2}(\text{sum in } T_L) + \frac{1}{2}(\text{sum in } T_R) \le 1.$$

This proves the theorem for Case 1.

3.1.2. Case 2. Suppose that one of the two spines has length less than $p+1$. Without loss of generality, assume that the left spine has length $l \le p$. Let T_R be the subtree rooted at the node at distance $p+1$ from the root along the right spine. By Lemma 3.1, it follows that the interval of uncertainty at the root of T_R, $[A_R, Z)$, and the one at the leaf at the end of the left spine, $[A, A+k+1)$, will overlap. In fact, if $[A, x_0, x_1, \ldots, x_{2p}, Z]$ is the distribution of probes in the interval of uncertainty at the root of the subtree, then $A+k+1 \ge x_{p+1}$ and $A_R \le x_p$. So, we can consider the term corresponding to the interval $[A+k, A+k+1)$ to be part of the sum over T_R. Thus,

$$\begin{aligned}\sum_{i=A}^{Z-1} \frac{1}{L_{c(i)+K}} &\le \sum_{i=A}^{A+k-1} \frac{1}{L_{l+k+L^{-1}(k)+1}} + \sum_{i=A_R}^{Z-1} \frac{1}{L_{c_R(i)+p+1+K}} \\ &\le \frac{k}{2k} + \frac{1}{2} \\ &= 1,\end{aligned}$$

completing the proof of the theorem for even k.

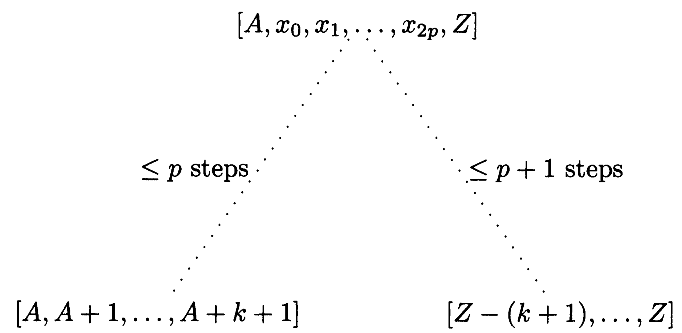


FIG. 6. *Base case for the inductive proof for odd k.*

3.2. k odd. Let $k = 2p+1$; we prove the following lemma, from which the theorem follows as a corollary.

LEMMA 3.2. *In a k-modal search tree, k odd, let $[A, x_0, x_1, \ldots, x_{k-1}, Z]$ be a node at distance l from the root and let q be a value in the range $A \leq q < Z$ having the least cost in the tree—that is, for all i, $A \leq i < Z$, $c(q) \leq c(i)$. Then*

$$\sum_{i=A}^{Z-1} \frac{1}{L_{c(i)+K}} \leq \frac{L_{c(q)+K-l}}{L_{c(q)+K}},$$

where L_n and K are defined by equations (3) *and* (4) *in Theorem* 2.5.

The following identity is used in the proof of Lemma 3.2 in the two subsections below. Its proof and that of some other identities describing the behavior of L_n when k is odd are given in the Appendix. For $i \geq 0$, $l > 0$, $j > 0$, if $i \equiv 2, 3, \ldots, p \pmod{p+1}$, then

$$(5) \qquad \frac{L_i}{L_{i+j}} = \frac{L_{i+l}}{L_{i+j+l}};$$

if $i \equiv 0$ or $1 \pmod{p+1}$ and $\lfloor \frac{i}{p+1} \rfloor$ is odd, then

$$(6) \qquad \frac{L_i}{L_{i+j}} > \frac{L_{i+l}}{L_{i+j+l}};$$

if $i \equiv 0$ or $1 \pmod{p+1}$ and $\lfloor \frac{i}{p+1} \rfloor$ is even, then

$$(7) \qquad \frac{L_i}{L_{i+j}} < \frac{L_{i+l}}{L_{i+j+l}}.$$

The proof of Lemma 3.2 is by induction on the height of the subtree. As in the proof of the theorem for even k, this proof uses Lemma 3.1 to infer that the intervals of uncertainty of the nodes at distance $p+1$ and $p+2$ down the left and right spines overlap and cover the entire interval of uncertainty at the root. This case, however, is much more complicated than for even k because the recurrence governing the lower bound does not have a simple, closed-form solution.

For the base case, the length of the left and right spines of the subtree are at most p and $p+1$ (see Fig. 6). Again, by Lemma 3.1, we know that the intervals of uncertainty at the leaves at the end of the left and right spine overlap and have at least two common segments. Hence, the length of the interval of uncertainty at the root is at most $2k$. So,

$$\sum_{i=A}^{Z-1} \frac{1}{L_{c(i)+K}} \leq \sum_{i=A}^{Z-1} \frac{1}{L_{c(q)+K}} \leq \frac{2k}{L_{c(q)+K}}.$$

Since $c(q) \geq l+k$, $L_{c(q)-l+K} \geq L_{k+L^{-1}(k)+1} \geq 2k$,

$$\sum_{i=A}^{Z-1} \frac{1}{L_{c(i)+K}} \leq \frac{L_{c(q)-l+K}}{L_{c(q)+K}}.$$

We now assume that the lemma is true for all subtrees of k-modal search trees of height at most h for any $h \geq p+1$.

For the induction step, we have a k-modal search tree in which both the spines are of length at least $p+1$. In the following discussion, we will assume that the interval of least cost, $[q, q+1)$, is contained in the subtree T_R rooted at a node at distance at most $p+2$ along the right spine. The case when $[q, q+1)$ lies in the left subtree is symmetric.

Let T_L and T_R be the subtrees rooted at distance $\min\{p+1, d_L\}$ and $\min\{p+2, d_R\}$ along the left and right spines, respectively, where d_L and d_R are the lengths of the left and right spines. From Lemma 3.1, we know that the intervals of uncertainty at the roots of T_L and T_R overlap.

3.2.1. Case 1. Suppose that the left and right spines have length at least $p+1$ and $p+2$, respectively; see Fig. 7. Let $[r, r+1)$ be the least-cost interval in T_L. By the induction hypothesis, we have

$$\text{sum in } T_L = \sum_{i=A}^{Z_L-1} \frac{1}{L_{c(i)+K}} \leq \frac{L_{c(r)-(l+p+1)+K}}{L_{c(r)+K}}$$

and

$$\text{sum in } T_R = \sum_{i=A_R}^{Z-1} \frac{1}{L_{c(i)+K}} \leq \frac{L_{c(q)-(l+p+2)+K}}{L_{c(q)+K}}.$$

Now, if

$$\frac{L_{c(r)-(l+p+1)+K}}{L_{c(r)+K}} \leq \frac{L_{c(q)-(l+p+1)+K}}{L_{c(q)+K}}, \tag{8}$$

then

$$\begin{aligned}
\sum_{i=A}^{Z-1} \frac{1}{L_{c(i)+K}} &\leq (\text{sum in } T_L) + (\text{sum in } T_R) \\
&\leq \frac{L_{c(r)-(l+p+1)+K}}{L_{c(r)+K}} + \frac{L_{c(q)-(l+p+2)+K}}{L_{c(q)+K}} \\
&\leq \frac{L_{c(q)-(l+p+1)+K}}{L_{c(q)+K}} + \frac{L_{c(q)-(l+p+2)+K}}{L_{c(q)+K}} \\
&= \frac{L_{c(q)-(l+p+1)+K} + L_{c(q)-(l+p+2)+K}}{L_{c(q)+K}} \\
&= \frac{L_{c(q)-l+K}}{L_{c(q)+K}}.
\end{aligned}$$

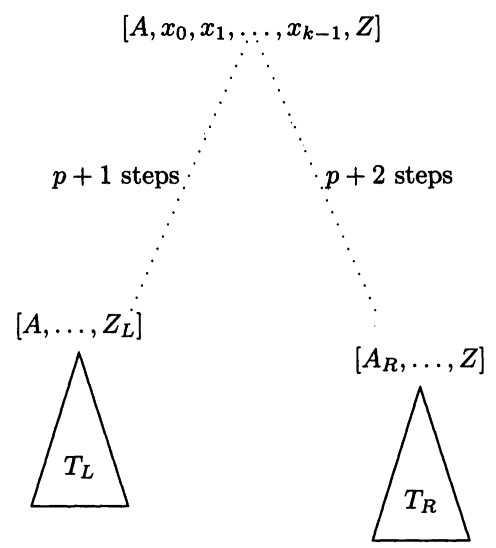


FIG. 7. *Induction when k is odd and no expansion of T_L is required.*

From (5) and (6), inequality (8) is true when $c(q) - (l + p + 1) + K \equiv 2, 3, \ldots, p \pmod{p+1}$ or when $c(q) - (l + p + 1) + K \equiv 0, 1 \pmod{p+1}$ and $\lfloor \frac{c(q)-(l+p+1)+K}{p+1} \rfloor$ is odd. If inequality (8) is not satisfied, then we have two subcases.

Case 1(a). Suppose the left and right spines of T_L have length at least $p+1$ and $p+2$, respectively. Let T_{LL} and T_{LR} be the subtrees at distance $p+1$ and $p+2$ along the left and right spine of T_L (see Fig. 8). Let $[s, s+1)$ and $[r, r+1)$ be the intervals of least cost in T_{LL} and T_{LR}, respectively. Then by the induction hypothesis,

$$\text{sum in } T_{LL} = \sum_{i=A}^{Z_{LL}-1} \frac{1}{L_{c(i)+K}} \le \frac{L_{c(s)-(l+2p+2)+K}}{L_{c(s)+K}},$$

$$\text{sum in } T_{LR} = \sum_{i=A_{LR}}^{Z_L-1} \frac{1}{L_{c(i)+K}} \le \frac{L_{c(r)-(l+2p+3)+K}}{L_{c(r)+K}},$$

and

$$\text{sum in } T_R = \sum_{i=A_R}^{Z-1} \frac{1}{L_{c(i)+K}} \le \frac{L_{c(q)-(l+p+2)+K}}{L_{c(q)+K}}.$$

Since inequality (8) is not satisfied,

$$a = \left\lfloor \frac{c(q) - (l + p + 1) + K}{p+1} \right\rfloor$$

is even. Hence,

$$\left\lfloor \frac{c(q) - (l + 2p + 2) + K}{p+1} \right\rfloor = \left\lfloor \frac{c(q) - (l + 2p + 3) + K}{p+1} \right\rfloor = a - 1$$

is odd. Thus, from inequality (6) it follows that

$$\frac{L_{c(s)-(l+2p+2)+K}}{L_{c(s)+K}} \le \frac{L_{c(q)-(l+2p+2)+K}}{L_{c(q)+K}}$$

and

$$\frac{L_{c(r)-(l+2p+3)+K}}{L_{c(r)+K}} \le \frac{L_{c(q)-(l+2p+3)+K}}{L_{c(q)+K}}.$$

Now, by Lemma 3.1,

$$\begin{aligned}\sum_{i=A}^{Z-1} \frac{1}{L_{c(i)+K}} &\le (\text{sum in } T_{LL}) + (\text{sum in } T_{LR}) + (\text{sum in } T_R)\\ &\le \sum_{i=A}^{Z_{LL}-1} \frac{1}{L_{c(i)+K}} + \sum_{i=A_{LR}}^{Z_L-1} \frac{1}{L_{c(i)+K}} + \sum_{i=A_R}^{Z-1} \frac{1}{L_{c(i)+K}}\\ &\le \frac{L_{c(s)-(l+2p+2)+K}}{L_{c(s)+K}} + \frac{L_{c(r)-(l+2p+3)+K}}{L_{c(r)+K}} + \frac{L_{c(q)-(l+p+2)+K}}{L_{c(q)+K}}\\ &\le \frac{L_{c(q)-(l+2p+2)+K}}{L_{c(q)+K}} + \frac{L_{c(q)-(l+2p+3)+K}}{L_{c(q)+K}} + \frac{L_{c(q)-(l+p+2)+K}}{L_{c(q)+K}}\\ &= \frac{L_{c(q)-l+K}}{L_{c(q)+K}}.\end{aligned}$$

Case 1(b). Suppose that T_L is not big enough to be sufficiently expanded. Let $y = \min\{d_{LL}, p+1\}$ and $z = \min\{d_{LR}, p+2\}$, where d_{LL} and d_{LR} are the lengths of the left and right spines of T_L, respectively.

If $y \le p$ and $z \le p+1$, then by Lemma 3.1, the intervals $[A, Z_{LL})$ and $[A_{LR}, Z_L)$ overlap and $A_{LR} - Z_{LL} \ge 2$. Since both these intervals correspond to leaves in the k-modal search tree, their length is $k+1$. So, the length of $[A, Z_L)$ is at most $2(k+1)-2 = 2k$. Furthermore, since $c(q) \le c(i)$ for any i such that $[i, i+1) \subset [A, Z_L)$, we have

$$\text{sum in } T_L = \sum_{i=A}^{Z_L-1} \frac{1}{L_{c(i)+K}} \le \frac{2k}{L_{c(q)+K}}.$$

Since $c(q) \ge k+l+1$,

$$L_{c(q)-(l+p+1)+K} \ge L_{k+L^{-1}(k)+1} \ge 2k.$$

So,

$$\text{sum in } T_L \le \frac{L_{c(q)-(l+p+1)+K}}{L_{c(q)+K}}.$$

By the induction hypothesis, we know that

$$\text{sum in } T_R \le \frac{L_{c(q)-(l+p+2)+K}}{L_{c(q)+K}}.$$

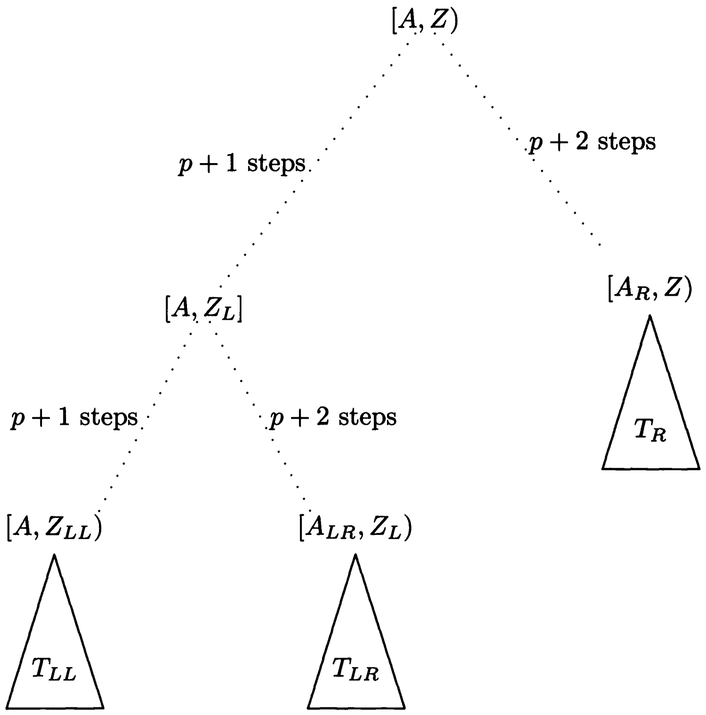


FIG. 8. *Subcase when T_L expands sufficiently.*

Hence,

$$\begin{aligned}\sum_{i=A}^{Z-1} \frac{1}{L_{c(i)+K}} &= (\text{sum in } T_L) + (\text{sum in } T_R) \\ &\leq \frac{L_{c(q)-l+K}}{L_{c(q)+K}}.\end{aligned}$$

If $y = p+1$ and $z \leq p+1$, then by Lemma 3.1, we know that $A_{LR} - Z_{LL} \geq 1$. Furthermore, since the interval $[A_{LR}, Z_L)$ corresponds to a leaf, we have

$$\text{sum in } T_L \leq (\text{sum in } T_{LL}) + \sum_{i=Z_L-k}^{Z_L-1} \frac{1}{L_{c(i)+K}}. \tag{9}$$

By the induction hypothesis,

$$\text{sum in } T_{LL} \leq \frac{L_{c(s)-(l+2p+2)+K}}{L_{c(s)+K}}. \tag{10}$$

By the same argument as in Case 1(a),

$$\frac{L_{c(s)-(l+p+1)+K}}{L_{c(s)+K}} \leq \frac{L_{c(q)-(l+p+1)+K}}{L_{c(q)+K}}.$$

Also, since $c(q) \leq c(i)$ for any i in the range $[A_{LR}, Z_L)$,

$$\sum_{i=Z_L-k}^{Z_L-1} \frac{1}{L_{c(i)+K}} \leq \frac{k}{L_{c(q)+K}}.$$

Since $c(q) \geq k + l + 1$,

$$L_{c(q)-(l+2p+3)+K} \geq L_{k-2p-2+L^{-1}(k)+1} = L_{L^{-1}(k)} = k.$$

So,

$$\sum_{i=Z_L-k}^{Z_L-1} \frac{1}{L_{c(i)+K}} \leq \frac{L_{c(q)-(l+2p+3)+K}}{L_{c(q)+K}}. \tag{11}$$

Thus by (9), (10), and (11),

$$\text{sum in } T_L \leq \frac{L_{c(q)-(l+p+1)+K}}{L_{c(q)+K}},$$

and by the induction hypothesis,

$$\text{sum in } T_R \leq \frac{L_{c(q)-(l+p+2)+K}}{L_{c(q)+K}},$$

yielding

$$\sum_{i=A}^{Z-1} \frac{1}{L_{c(i)+K}} = (\text{sum in } T_L) + (\text{sum in } T_R) \leq \frac{L_{c(q)-l+K}}{L_{c(q)+K}}$$

as desired.

The case when $y \leq p$ and $z = p + 2$ is identical to the above, *mutatis mutandis*, completing the proof of Lemma 3.2 for Case 1.

3.2.2. Case 2. Suppose that the spines of the k-modal search tree are not sufficiently long. Again, we consider two subcases.

Case 2(a). Assume that the left spine has length $y \leq p$ and the right spine has length at least $p+2$. Let T_R be the subtree rooted at the node at distance $p+2$ along the right spine (see Fig. 9). By Lemma 3.1, we know that

$$\sum_{i=A}^{Z-1} \frac{1}{L_{c(i)+K}} \leq \sum_{i=A}^{A+k} \frac{1}{L_{c(i)+K}} + (\text{sum in } T_R).$$

By the induction hypothesis,

$$\text{sum in } T_R \leq \frac{L_{c(q)-(l+p+2)+K}}{L_{c(q)+K}}.$$

So,

$$\begin{aligned}\sum_{i=A}^{Z-1} \frac{1}{L_{c(i)+K}} &\leq \sum_{i=A}^{A+k} \frac{1}{L_{c(i)+K}} + \frac{L_{c(q)-(l+p+2)+K}}{L_{c(q)+K}} \\ &\leq \frac{k+1}{L_{c(q)+K}} + \frac{L_{c(q)-(l+p+2)+K}}{L_{c(q)+K}}.\end{aligned}$$

Since $c(q) \geq l + k + 1$,

$$L_{c(q)-(l+p+1)+K} \geq L_{k+L^{-1}(k)+1} \geq 2k \geq k+1.$$

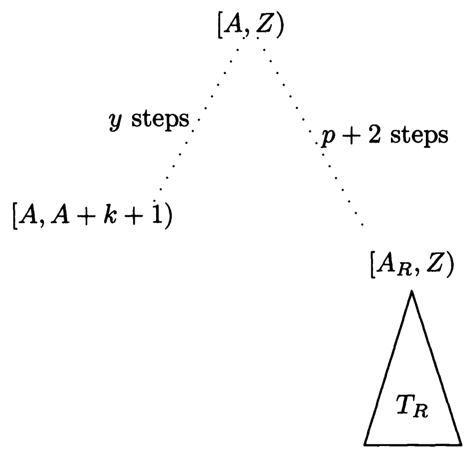


FIG. 9. *Induction when T_L does not expand sufficiently.*

Hence,

$$\sum_{i=A}^{Z-1} \frac{1}{L_{c(i)+K}} \le \frac{L_{c(q)-(l+p+1)+K}}{L_{c(q)+K}} + \frac{L_{c(q)-(l+p+2)+K}}{L_{c(q)+K}}$$
$$= \frac{L_{c(q)-l+K}}{L_{c(q)+K}}$$

as desired.

Case 2(b). Suppose the right spine has length $z \le p + 1$ and the left spine has length at least $p + 1$. This case is symmetric to Case 2(a). To show that the sum in T_L is sufficiently small, we use exactly the same arguments as in Case 1.

This completes the proof of Lemma 3.2, thus proving Theorem 2.5.

4. Algorithms for *k*-modal search. We first note that an algorithm for $(k-1)$-modal search can be used for k-modal searching because the derivative of a k-modal function f is the $(k-1)$-modal function Δf. Hence, we can replace each probe of Δf by the algorithm for $(k-1)$-modal searching by two probes at adjacent points, and use Δf as the result of the probe made by the algorithm for $(k-1)$-modal search. This gives an algorithm for k-modal searching that uses twice as many probes as the algorithm for $(k-1)$-modal searching. However, these naïve algorithms are not optimal. The key to designing optimal algorithms for k-modal searching is to find a probing strategy that insures that the k probes in the current interval of uncertainty divide that interval in such a way that both the outermost segments are "sufficiently large." Since this property must be satisfied recursively by the configurations that result after one of the outermost segments has been eliminated, it is necessary that the distribution of probes be "balanced" and the probing strategy must regenerate a distribution of probes similar to the starting distribution.

4.1. Optimal algorithms for even *k*. When $k = 2p$, the recurrence relation used in deriving the lower bound on the number of probes required is

$$L_n = 2L_{n-(p+1)},$$

with $L_i = 1$ for $0 \le i \le p$. The algorithm halves the length of the interval of uncertainty after $p + 1$ probes and is hence optimal with respect to the number of

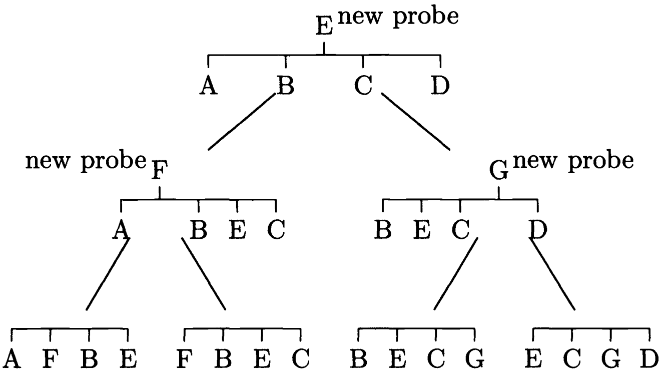


FIG. 10. *Part of the bimodal search tree generated by the optimal bimodal search algorithm.*

probes used. This algorithm is an extension of the discrete case of the algorithm proposed in [7].

Initially, the algorithm distributes the k probes uniformly over the interval $[0, N)$. Thus, the configuration of probes in the initial interval of uncertainty is of the form $[0, L_m, 2L_m, 3L_m, \ldots, 2pL_m, N]$, where m is chosen so that the rightmost segment has length at most L_m, $m = L^{-1}(N/(k+1))$. Now, the algorithm probes at the midpoint of the unique center segment—this is the $(p+1)$st segment from either end of the interval of uncertainty—dividing it into two segments of length $L_{m-(p+1)}$ each. Next, either the leftmost or the rightmost segment is discarded and a new probe is made at the midpoint of the segment adjacent to the center segment on the side *opposite* the side that lost a segment. The algorithm maintains a central "region of division" consisting of segments half as long as the segments on either side of this region. A new probe is always made at the midpoint of a segment adjacent to the "region of division" on the side opposite the side that lost a segment on the previous probe. This guarantees that the first p probes cause segments of length L_m to be discarded (except the first probe, which might result in the loss of the rightmost segment, which might have length less than L_m). After $p+1$ probes, the algorithm has a configuration of probes that divides the interval of uncertainty into segments of length $L_{m-(p+1)}$, except the rightmost segment, which can be shorter. Thus, after $p+1$ probes, the interval of uncertainty is halved and a configuration similar to the starting one is regenerated; this process is repeated until the interval of uncertainty is of length $k+1$. A part of a bimodal search tree generated by the above algorithm is shown in Fig. 10.

In the pseudocode of Algorithm 4.1, we use a binary search tree to store the pairs $(x, f(x))$ corresponding to probes in the current interval of uncertainty. The function *InsertPair* inserts a new pair $(x, f(x))$ into the binary search tree. *DeleteExtremeSegment* deletes the pair corresponding to the right or left endpoint of the interval of uncertainty. *FindSide* determines which of the two extreme segments in the current interval of uncertainty can be discarded by computing the sign of $P(x_0, x_1, \ldots, x_k, f)$, defined in the proof of Lemma 2.3 (see Fig. 2). Notice that our time complexity measure considers only function evaluations, ignoring the $O(\log k)$-time-per-tree operation.

4.2. Optimal algorithm for trimodal search. Goldstein and Reingold [6] presented an optimal algorithm for discrete unimodal ($k = 1$) search based on Kiefer's

ALGORITHM 1. *Optimal discrete k-modal search over the interval $[0, N)$.*

```
function ModalSearch(
    f: function(NonNegativeInteger): real; {k-modal function to be searched }
    k: NonNegativeInteger; {Modality of function f}
    low: NonNegativeInteger;
    high: PositiveInteger; {Interval [low, high) defines the domain of f}
    ): NonNegativeInteger; { Left endpoint of an interval of length $k+1$
                              containing the unique zero of $\Delta^k f$ }
{ Search over interval [low, high), $high - low \geq k+2$ }
procedure InsertPair(
    Fvalues: BinarySearchTree;
    x: NonNegativeInteger);
{Evaluates $f(x)$ and inserts the pair $(x, f(x))$ into a binary search tree indexed by x }
    ⋮
end;
procedure Search
    ⋮ { See Algorithm 4.1 }
end;
var
    i: NonNegativeInteger; { Left endpoint of interval containing the unique zero of $\Delta^k f$ }
    j, m: NonNegativeInteger;
    Fvalues: BinarySearchTree;
    p: NonNegativeInteger;

begin
    $p := \lfloor k/2 \rfloor$;
    $m := L^{-1}((high - low)/k + 1)$;
    for j = 1 to k do
        InsertPair(Fvalues, $low + jL_m$);
    i := low;
    Search(i, high, m);
    ModalSearch := i
end
```

Fibonacci search algorithm for the continuous version of unimodal search [10]. In this section, we give an optimal algorithm for trimodal search, modeled after Hyafil [7], and in the next section, we sketch a family of similar but not quite optimal algorithms for odd $k > 3$.

For trimodal searching, the recurrence relation governing the lower bound on the number of probes needed is

$$L_n = L_{n-2} + L_{n-3}. \tag{12}$$

It is easy to show that we also have

$$L_n = L_{n-1} + L_{n-5}. \tag{13}$$

For simplicity, assume that the initial length of the interval of uncertainty is L_n. Then, equations (12) and (13) can be used to generate an initial distribution of probes in

ALGORITHM 2. *Discrete k-modal search procedure.*

```
procedure Search(
    var i: NonNegativeInteger;{Upon entry, search has reached ...}
    j, m: NonNegativeInteger); { ...node $[i, i + L_m, i + 2L_m, \ldots, i + kL_m, j]$ }
procedure DeleteExtremeSegment(
    Fvalues: BinarySearchTree;
    side: (LEFT, RIGHT) );
{Deletes the pair $(x, f(x))$, corresponding to the left or the right
endpoint of the interval of uncertainty, from Fvalues }
    ⋮
end;
function FindSide( Fvalues: BinarySearchTree): (LEFT, RIGHT);
{This function determines the side that will lose its extreme segment by applying
Lemma 2.4, after $k + 1$ probes have been made}
    ⋮
end;
var
    left, right: NonNegativeInteger; {These delimit the left and right boundaries
                                      of the "region of division"}
    Side: (LEFT, RIGHT); {Flag indicating which side lost a segment as a result
                          of the last probe}
    num: NonNegativeInteger;
    NewProbe: NonNegativeInteger;
begin {Search}
  if $j - i \le k + 1$ then
    {$k$-modal search ends at interval $[i, i + k + 1)$}
  else begin
    left := right := $i + (p + 1)L_m$;
    Side := RIGHT;

    for num := 1 to $p + 1$ do begin
      if Side = LEFT then begin {Probe just beyond the right end of the
                                 "region of division"}
        NewProbe := right $+L_{m-(p+1)}$;
        right := right $+L_m$
        end
      else begin {Probe just to the left of the left end of the "region of division"}
        NewProbe := left $-L_{m-(p+1)}$;
        left := left $-L_m$
        end
      InsertPair(Fvalues, NewProbe);
      Side := FindSide(Fvalues);
      DeleteExtremeSegment(Fvalues, Side)
      end {of for loop}

    { Now we make the recursive calls }
    if Side = LEFT then begin
      i := left $+L_{m-(p+1)}$;
      Search(i, right, $m - (p + 1)$)
      end
    else begin
      i := left;
      Search(i, right $-L_{m-(p+1)}$, $m - (p + 1)$)
      end
    end {of else}
end
```

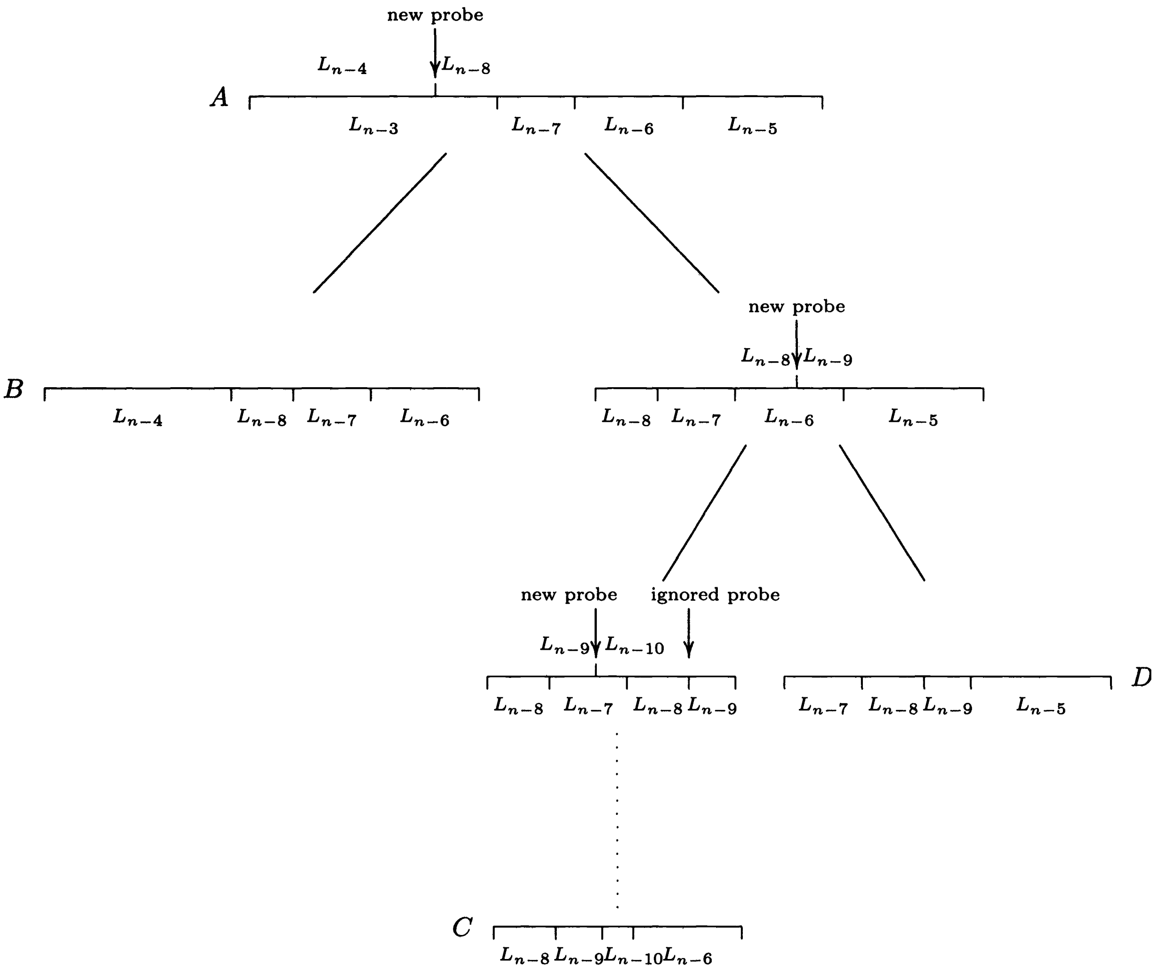


FIG. 11. *Pattern of probes in the optimal trimodal search algorithm. The root interval, labeled A, has length $L_n = L_{n-3} + L_{n-7} + L_{n-6} + L_{n-5}$; its left child, labeled B, has length $L_{n-1} = L_{n-4}+L_{n-8}+L_{n-7}+L_{n-6}$. The interval labeled C has length $L_{n-3} = L_{n-8}+L_{n-9}+L_{n-10}+L_{n-6}$. The interval labeled D has length $L_{n-2} = L_{n-7} + L_{n-8} + L_{n-9} + L_{n-5}$.*

which the segment lengths are, respectively, $L_{n-3}, L_{n-7}, L_{n-6}, L_{n-5}$. The subsequent pattern of probes in the optimal trimodal search algorithm is shown in the part of the trimodal search tree in Fig. 11. The distribution of probes at the leaves of this subtree is the same (up to symmetry) as the initial distribution of probes. This enables the recursive application of this probing strategy at these leaves. Furthermore, at leaf B the length of the interval of uncertainty is L_{n-1} after one probe, at leaf D the length of the interval of uncertainty is L_{n-2} after two probes, and at leaf C the length of the interval of uncertainty is L_{n-3} after three probes, insuring optimality. Notice that to regenerate a distribution of probes symmetric to the initial distribution of probes at leaf C, a "rearrangement probe" is made and an earlier probe is ignored.

4.3. Near-optimal algorithms for odd $k \geq 5$. Suppose $k = 2p + 1$, $p > 1$. For simplicity, assume that the length of the initial interval of uncertainty is L_n, where L_n is a member of the series defined by recurrence (3). To get the initial

ALGORITHM 3. *Algorithm for finding all local extrema of a k-modal function.*

```
var
    extrema: array of NonNegativeInteger; {stores the extrema found so far}
procedure FindAllExtrema(
    f: function(NonNegativeInteger): real;
    k: NonNegativeInteger; {Modality of function f}
    low: NonNegativeInteger;
    high: PositiveInteger); {f is k-modal in the range [low, high)}
procedure InsertExtremum(i: NonNegativeInteger);
{Inserts i in the array extrema, indicating an extremum in [i, i + k + 1)}
    ⋮
end;
var
    i: NonNegativeInteger;
begin
    i := ModalSearch(k, low, high, f);
    if k = 1 then
        InsertExtremum(i);
    else begin
        FindAllExtrema(f, k − 1, low, i + k + 1);
        FindAllExtrema(f, k − 1, i + 1, high)
        end
end
```

distribution of probes, we recursively divide the largest segment in the current set of segments into two parts, using recurrence (3), until there are $k+1$ segments. This results in an initial configuration of probes that has all segments with lengths in the set $\{L_{n-K_1}, L_{n-K_1+1}, \dots, L_{n-K_2}\}$, where K_1 and K_2 are constants depending on p. Now, the algorithm uses the same strategy as in the case when k is even; starting with one of the two center segments, a probe is made in a segment adjacent to the "region of division" on the side opposite the side that lost a segment as a result of the previous probe. Unfortunately, the nonuniqueness of the center segment (since $k+1$ is even) forces the algorithm to use, in the worst case, $p+2$ probes to shift the indices of the set of lengths of segments down by $p+1$. Thus, the algorithm is suboptimal, but as the value of k increases, its performance approaches the lower bound since the $\frac{p+1}{p+2} \to 1$ as $p \to \infty$. The proper choice of the distribution of segments of different lengths in the interval of uncertainty might hold the key to designing optimal algorithms for these cases.

5. Finding all extrema of a k-modal function.

We can use a k-modal search to obtain an efficient algorithm for finding all the local extrema (or zeros) of a k-modal function. The algorithm makes use of the following lemma.

LEMMA 5.1. *Let $f : \{0, \dots, N\} \to \Re$ be a k-modal function and let $[i, i+1)$ be the interval containing the unique zero of its kth derivative. Then the functions $f : \{0, \dots, i+k+1\} \to \Re$ and $f : \{i+1, \dots, N\} \to \Re$ are both $(k-1)$-modal.*

Proof. Since $[i, i+1)$ is the unique zero of $\Delta^k f$, Δ^{k-1} is necessarily unimodal. Assume without loss of generality that the part of $\Delta^{k-1} f$ to the left of $i+1$ is decreasing and the part to the right is increasing. We now note that $\Delta^{k-1} f$ in the range $[0, i+1)$ is determined by the values of f in the range $[0, i+k)$. Also, the values of $\Delta^{k-1}f$ in the range $[i+1, N-k-1)$ are determined by the values of f in the range $[i+1, N)$. Now, when splitting the original k-modal function f into the ranges $[0, i+k)$ and $[i+1, N)$, we will assume that $f(i+k) = -\infty$ for the function f over the range $[0, i+k)$ and $f(i+1) = \infty$ for the restriction of f to the range $[i+1, N)$. This then implies that both the restrictions of f defined above have a unique zero in their $(k-1)$st derivative and are hence $(k-1)$-modal. □

This lemma suggests a divide-and-conquer algorithm for finding all the local extrema (or zeros) of a k-modal function in which k-modal searching serves as the "divide" step. It is important to note that the number of local extrema of f need not be symmetrically distributed about the unique zero of the kth derivative of the k-modal function. In fact, there might be just one local extremum of f on one side and all the others (at most $k-1$) might lie on the other side.

A divide-and-conquer algorithm based on Lemma 5.1 is presented as Algorithm 3.

The k-modal searching algorithms discussed in §4 use at most $(\lceil k/2 \rceil + 1)\lg N$ probes for a k-modal function over the range $[0, N)$. The recurrence relation for the number of probes made in the worst case of Algorithm 4.3 is of the form

$$T(N,k) = \max_{0 \le i < N} \{T(i+k+1, k-1) + T(N-i-1, k-1)\} + (\lceil k/2 \rceil + 1)\lg N.$$

Furthermore, in this recurrence, the worst case arises when the division is perfectly balanced. Hence,

$$T(N,k) \le 2T(N/2, k-1) + (\lceil k/2 \rceil + 1)\lg N. \tag{14}$$

On unfolding this recurrence, we get

$$\begin{aligned} T(N,k) &\le \sum_{i=0}^{\min\{k,\lg N\}} (\lceil (k-i)/2 \rceil + 1)2^i \lg(N/2^i) \\ &< k \sum_{i=0}^{\min\{k,\lg N\}} 2^i \lg N \\ &= O(k2^{\min\{k,\lg N\}} \lg N). \end{aligned}$$

Thus the algorithm is better than the naïve sequential scan algorithm when $k < \lg N - 2\lg\lg N$. But, this algorithm does not make use of the probes made during k-modal searching while doing $(k-1)$-modal searching; hence, it should be possible to devise more efficient algorithms for finding all the local extrema of a k-modal function by reusing some of the probes made in k-modal search during the subsequent lower modal searches.

6. Unbounded k-modal search. In the unbounded discrete k-modal search problem, the search takes place over an infinite domain, the nonnegative integers. The problem of unbounded search was first introduced by Bentley and Yao [3]. Reingold and Shen [17] presented a hierarchy of successively better algorithms and corresponding lower bounds. Goldstein and Reingold [6] extended some of these ideas to unimodal searching.

6.1. Algorithms for unbounded discrete k-modal search. The infinite sequence of increasingly better algorithms for unbounded searching described for monotonic functions [17] and for unimodal functions [6] can be extended to k-modal searching. This follows from the observation that Lemma 2.4 gives us an efficient way of finding a finite interval containing the unique zero of the k-modal function by maintaining a set of $k+2$ probes and making probes at successive values of an Ackermann-like (rapidly growing) function. By computing the sign of the P function (see Lemma 2.4) for the last $k+1$ probes in the current set of probes, we can decide whether the zero lies in the interval spanned by the current set of probes. This strategy is used to mimic the top level search in [17] and [6]. The level-by-level search requires the definition of a hierarchy of Ackermann-like functions, $A_l(n)$, satisfying

$$A_l(n) = A_{l-1}(A_l(n-1)).$$

The function $A_1(n)$ needs to be chosen carefully so that the distribution of probes when the search reaches level $l = 1$ conforms to the distribution required by the finite k-modal search algorithm. The exact details of these Ackermann-like functions have been a stumbling block in our exploration of unbounded k-modal search.

6.2. Lower bounds for unbounded discrete k-modal search. The main tool used in generating lower bounds for unbounded discrete k-modal searching is the following infinite version of the generalized Kraft's inequality (2), which is a corollary of Theorem 2.5.

COROLLARY 6.1. *In an unbounded discrete k-modal search, if $c(i)$ is the number of probes used in the worst case when the unique zero of $\Delta^k f$ lies in $[i, i+1)$, then*

$$\sum_{i=0}^{\infty} \frac{1}{L_{c(i)+K}} \leq 1. \tag{15}$$

Proof. If not, then there must be some N for which

$$\sum_{i=0}^{N-1} \frac{1}{L_{c(i)+K}} > 1,$$

contradicting Theorem 2.5. □

As a direct consequence of Corollary 6.1, we have the following corollary that is used to prove lower bounds for unbounded discrete k-modal search.

COROLLARY 6.2. *Let $c(i)$ be the number of probes used in the worst case by an unbounded discrete k-modal search algorithm when the unique zero of $\Delta^k f$ occurs in the range $[i, i+1)$. If for some nondecreasing function d, the sum $\sum_{i=0}^{\infty} \frac{1}{L_{d(i)+K}}$ diverges, then $c(i) > d(i)$ for infinitely many i.*

Unfortunately, the algebra in trying to prove lower bounds using this approach gets too involved, at least for the definitions of the Ackermann-like functions that we examined.

7. Conclusions. One of the important problems related to k-modal searching is the the determination of the modality of a function. Although a linear scan can find all the turning points of the function, there does not seem to be an obvious linear-time algorithm for finding the modality of a function, when modality is defined as the largest derivative of the function having a unique zero. It would be interesting to find out if k-modal searching can be used to solve this problem.

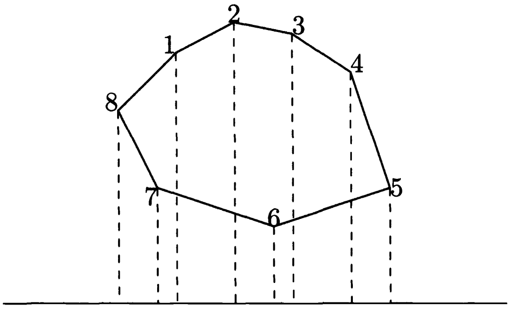


FIG. 12. *The distances of the vertices of a convex polygon from a line, when considered in order around the perimeter, yield a bimodal function.*

Bimodal search has been used in [4] to design algorithms for intersecting convex objects in two and three dimensions. These algorithms use the fact that the distances of the vertices of a convex polygon from a line yield a bimodal function (see Fig. 12). This raises the following question: given a polygon with a bounded number of concavities, do the distances of its vertices from a line yield a k-modal function? If so, can the algorithm for intersecting two-dimensional convex polygons be extended to an algorithm for intersecting nonconvex polygons with a bounded number of concavities by using k-modal search?

Using generalizations of Kraft's inequality to prove lower bounds seems to be applicable to several problems such that an algorithm for the problem can be mapped to a computation tree. A generalized Kraft's inequality then captures the minimum "degree of imbalance" that must be present in any computation tree solving the problem. In particular, the problem of discrete search in the presence of lies [16], [18], [21] might be a suitable candidate for a problem which can be attacked using this technique. It would also be interesting to see whether the lower-bound proof in this case suggests an optimal algorithm for discrete searching in the presence of lies, as was the case in k-modal searching.

Appendix: Some combinatorial identities. Here we give some identities describing the behavior of L_n when k is odd. Let $k = 2p+1$; then the recurrence relation defining the sequence L used in the theorem is

$$L_n = \begin{cases} 0 & \text{if } n = 0, \\ 1 & \text{if } 1 \le n \le p+1, \\ L_{n-(p+1)} + L_{n-(p+2)}, & \text{otherwise.} \end{cases}$$

When $p = 0$ this sequence is the Fibonacci sequence, so the identities we prove are generalizations of some well-known identities for the Fibonacci numbers.

PROPOSITION 7.1. *For $m \ge p+1$, $n \ge 0$,*

$$L_{m+n} = L_m L_{n+p+1} + L_{m-(p+1)} L_n.$$

Proof. We use induction on m and n. Assume that m is fixed. For the base case we have $n = 0$, so the proposition reduces to

$$L_m = L_m L_{p+1} + L_{m-(p+1)} L_0.$$

Since $L_{p+1} = 1$ and $L_0 = 0$, the proposition holds for the base case. Assume that the proposition holds for all $n \leq q$ and consider the case when $n = q + 1$:

$$L_{m+q+1} = L_{m+(q-p)} + L_{m+(q-p-1)}.$$

Applying the induction hypothesis to both of the terms on the right-hand side, we have

$$L_{m+q+1} = L_m L_{q+1} + L_{m-(p+1)} L_{q-p} + L_m L_q + L_{m-(p+1)} L_{q-p-1},$$

which, on collecting the common terms, reduces to

$$L_{m+q+1} = L_m L_{p+q+2} + L_{m-(p+1)} L_{q+1}.$$

This proves the proposition for the case $n = p + 1$, thus completing the proof by induction. A similar proof works when we assume n is fixed. □

PROPOSITION 7.2. *For $n \geq p+1$,*

$$L_{n+(p+1)} L_{n-(p+1)} - L_n^2 = \begin{cases} (-1)^{\lfloor \frac{n}{p+1} \rfloor} & \textit{if } n \equiv 0 \textit{ or } 1 \pmod{p+1}, \\ 0 & \textit{otherwise.} \end{cases}$$

Proof. The proof is by induction on n. For the base case, $n = p + 1$ and the identity is obvious. The induction step uses the following matrix identity, which is a consequence of Proposition 7.1:

$$\begin{bmatrix} L_{n+p+1} & L_n \\ L_n & L_{n-(p+1)} \end{bmatrix} = \begin{bmatrix} L_{2(p+1)} & L_{p+1} \\ L_{p+1} & 0 \end{bmatrix} \begin{bmatrix} L_n & L_{n-(p+1)} \\ L_{n-(p+1)} & L_{n-2(p+1)} \end{bmatrix}.$$

Taking determinants on both sides of this identity, we get

$$L_{n+p+1} L_{n-(p+1)} - L_n^2 = -L_{p+1}^2 (L_n L_{n-2(p+1)} - L_{n-(p+1)}^2).$$

Since $L_{p+1} = 1$ and by the induction hypothesis

$$L_n L_{n-2(p+1)} - L_{n-(p+1)}^2 = \begin{cases} (-1)^{\lfloor \frac{n-(p+1)}{p+1} \rfloor} & \text{if } n - (p+1) \equiv 0 \text{ or } 1 \pmod{p+1}, \\ 0 & \text{otherwise,} \end{cases}$$

we get the required identity. □

PROPOSITION 7.3. *For $m, n \geq p+1$,*

$$L_{n+m-(p+1)} L_n - L_{n-(p+1)} L_{m+n} = \begin{cases} L_m (-1)^{\lfloor \frac{n}{p+1} \rfloor + 1} & \textit{if } n \equiv 0 \textit{ or } 1 \pmod{p+1}, \\ 0 & \textit{otherwise.} \end{cases}$$

Proof. Using Proposition 7.1 to expand the terms on the left-hand side of the identity, we have

$$\begin{aligned} &L_{n+m-(p+1)} L_n - L_{n-(p+1)} L_{m+n} \\ &= L_n (L_m L_n + L_{m-(p+1)} L_{n-(p+1)}) - L_{n-(p+1)} (L_m L_{n+p+1} + L_{m-(p+1)} L_n) \\ &= L_m (L_n^2 - L_{n-(p+1)} L_{n+p+1}). \end{aligned}$$

Now, using Proposition 7.2, we get the required identity. □

PROPOSITION 7.4. *For $i \geq 0$, $l > 0$, $j > 0$, if $i \equiv 2, 3, \ldots, p \pmod{p+1}$, then*

$$\frac{L_i}{L_{i+j}} = \frac{L_{i+l}}{L_{i+j+l}};$$

if $i \equiv 0$ or $1 \pmod{p+1}$ and $\lfloor \frac{i}{p+1} \rfloor$ is odd, then

$$\frac{L_i}{L_{i+j}} > \frac{L_{i+l}}{L_{i+j+l}};$$

if $i \equiv 0$ or $1 \pmod{p+1}$ and $\lfloor \frac{i}{p+1} \rfloor$ is even, then

$$\frac{L_i}{L_{i+j}} < \frac{L_{i+l}}{L_{i+j+l}}.$$

Proof. These realtions can be written jointly as

$$L_i L_{i+j+l} <^? > L_{i+l} L_{i+j},$$

which on expansion, using Proposition 7.1, yields

$$L_i(L_l L_{i+j+p+1} + L_{l-(p+1)} L_{i+j}) <^? > L_{i+j}(L_l L_{i+p+1} + L_i L_{l-(p+1)}).$$

Simplifying, we have

$$L_i L_{i+j+p+1} <^? > L_{i+j} L_{i+p+1}$$

or

$$L_i L_{i+j+p+1} - L_{i+j} L_{i+p+1} <^? > 0.$$

Using Proposition 7.3 with $m = j$ and $n = i + p + 1$, we get

$$\left\{ \begin{array}{ll} L_j(-1)^{\lfloor \frac{i}{p+1} \rfloor + 1} & \text{if } i \equiv 0 \text{ or } 1 \pmod{p+1}, \\ 0 & \text{otherwise.} \end{array} \right\} <^? > 0.$$

This yields the required identities. □

Acknowledgments. The authors would like to thank the referees for valuable comments and for pointing out references [13], [14], and [19]. The first author would like to thank Professor C. L. Liu for his support and encouragement during this work.

REFERENCES

[1] A. AGGARWAL AND R. C. MELVILLE, *Fast computation of the modality of polygons*, J. Algorithms, 7 (1986), pp. 369–381.

[2] M. AVRIEL AND D. J. WILDE, *Optimality proof for the symmetric Fibonacci search technique*, Fibonacci Quart., 4 (1966), pp. 265–269.

[3] J. L. BENTLEY AND A. C.-C. YAO, *An almost optimal algorithm for unbounded searching*, Inform. Process. Lett., 5 (1976), pp. 82–87.

[4] B. CHAZELLE AND D. P. DOBKIN, *Intersection of convex objects in two and three dimensions*, J. Assoc. Comput. Mach., 34 (1987), pp. 1–27.

[5] G. DAHLQUIST AND A. BJÖRCK, *Numerical Methods*, Prentice–Hall, Englewood Cliffs, NJ, 1974.

[6] A. S. GOLDSTEIN AND E. M. REINGOLD, *A Fibonacci version of Kraft's inequality applied to discrete unimodal search*, SIAM J. Comput., 22 (1993), pp. 751–777.

[7] L. HYAFIL, *Optimal search for the zero of the* $(n-1)^{st}$ *derivative*, Report de recherche 247, IRIA Laboria, Le Chesnay, France, 1977.

[8] R. M. KARP, *Minimum-redundancy coding for the discrete noiseless channel*, IRE Trans. Inform. Theory, 7 (1961), pp. 27–39.

[9] R. M. KARP AND W. L. MIRANKER, *Parallel minimax search for a maximum*, J. Combin. Theory, 4 (1968), pp. 19–35.

[10] J. KIEFER, *Sequential minimax search for a maximum*, Proc. Amer. Math. Soc., 4 (1953), pp. 502–505.

[11] L. G. KRAFT, *A device for quantizing, grouping and coding amplitude modified pulses*, Master's thesis, Electrical Engineering Department, Massachusetts Institute of Technology, Cambridge, MA, 1949.

[12] H. T. KUNG, *Synchronized and asynchronous parallel algorithms for multiprocessors*, in Algorithms and Complexity: New Directions and Recent Results, J. F. Traub, ed., Academic Press, New York, 1976, pp. 153–200.

[13] ———, *The complexity of obtaining starting points for solving operator equations by Newton's method*, in Analytic Computational Complexity, J. F. Traub, ed., Academic Press, New York 1976, pp. 35–57.

[14] E. NOVAK AND K. RITTER, *Some complexity results for zero finding for univariate functions*, J. Complexity, 9 (1993), pp. 15–40.

[15] L. T. OLIVER AND D. J. WILDE, *Symmetrical sequential minimax search for a maximum*, Fibonacci Quart., 2 (1964), pp. 169–175.

[16] A. PELC, *Solution of Ulam's problem on searching with a lie*, J. Combin. Theory, Ser. A, 44 (1987), pp. 129–140.

[17] E. M. REINGOLD AND X. SHEN, *More nearly optimal algorithms for unbounded searching, part I: The finite case*, SIAM J. Comput., 20 (1991), pp. 156–183.

[18] R. L. RIVEST, A. R. MEYER, D. J. KLIETMAN, K. WINKLMANN, AND J. SPENCER, *Coping with errors in binary search procedures*, J. Comput. System Sci., 20 (1980), pp. 396–404.

[19] K. SIKORSKI, *Optimal solution of nonlinear equations*, J. Complexity, 1 (1985), pp. 197–209.

[20] S. S. SKIENA, *Interactive reconstruction via geometric probing*, Proc. IEEE, 9 (1992), pp. 1364–1383.

[21] S. M. ULAM, *Adventures of a Mathematician,* Scribner's, New York 1976.

[22] B. S. VEROY, *An optimal algorithm for search of extrema of a bimodal function*, J. Complexity, 2 (1986), pp. 323–332.

[23] C. WITZGALL, *Fibonacci search with arbitrary first evaluation*, Fibonacci Quart., 10 (1972), pp. 113–134.

SIAM J. COMPUT.
Vol. 25, No. 2, pp. 448–476, April 1996

013

AN ALGEBRAIC MODEL FOR COMBINATORIAL PROBLEMS*

RICHARD E. STEARNS† AND HARRY B. HUNT III†

Abstract. A new algebraic model, called the generalized satisfiability problem (GSP) model, is introduced for representing and solving combinatorial problems. The GSP model is an alternative to the common method in the literature of representing such problems as language-recognition problems. In the GSP model, a problem instance is represented by a set of variables together with a set of terms, and the computational objective is to find a certain sum of products of terms over a commutative semiring. The model is general enough to express all the standard problems about sets of clauses and generalized clauses, all nonserial optimization problems, and all {0,1}-linear programming problems. The model can also describe many graph problems, often in a very direct structure-preserving way. Two important properties of the model are the following:

1. In the GSP model, one can naturally discuss the structure of individual problem instances. The structure of a GSP instance is displayed in a "structure tree." The smaller the "weighted depth" or "channelwidth" of the structure tree for a GSP instance, the faster the instance can be solved by any one of several generic algorithms.
2. The GSP model extends easily so as to apply to hierarchically specified problems and enables solutions to instances of such problems to be found directly from the specification rather than from the (often exponentially) larger specified object.

Key words. GSP, structure tree, SAT, separator, tree decomposition, treewidth, bounded bandwidth, nonserial optimization, hierarchical specifications

AMS subject classifications. 68Q25, 90C27

1. Introduction.

It has proven very useful to model computational problems as language-recognition problems. Under this paradigm, a computational problem is expressed as the question "Does an input string belong to a specified language?" This approach enables the application of concepts from automata theory including the concepts of P, NP, NP-completeness etc., [8, 15, 10]. In this paper, we develop a different model for computational problems, an algebraic model we call the GSP model. GSP is an abbreviation of "generalized satisfiability problem." In this model, a computational problem is expressed as the question "given a set of variables and terms using these variables, what is the value of a certain sum of products on these terms?" We call this collection of variables and terms a "formula."

The advantage of representing a problem instance as a formula is that problem instances then have a combinatorial structure determined by which variables appear in which terms. Using an appropriate (nonunique) representation of this structure, a certain kind of "subproblem independence" is displayed in a manner that can be exploited by any of several algorithms to speed up the computation of a formula's value.

We call the data structure used to display formula structure a "structure tree." We give several algorithms which take a formula and structure tree as input and compute the sum-of-products "value" of the formula. In these algorithms, the order of operations and the flow of control are determined exclusively by the problem structure as displayed in the structure tree. The operations performed and values manipulated are determined exclusively by the meaning of the terms and specific "sum" and "product" operators which define the formula value. The algorithms are

* Received by the editors January 19, 1993; accepted for publication (in revised form) September 7, 1994. This research was supported in part by National Science Foundation grant CCR-89-03313.

† Department of Computer Science, State University of New York at Albany, Albany, NY 12222.

"generic" in the sense that they can be described with Pascal-like pseudocode using uninterpreted operators and functions.

The complexity of the generic algorithms can be expressed in terms of the numbers of each kind of operation performed. These numbers depend primarily on the tree's "weighted depth" or "channelwidth" as defined in §3. Some of these algorithms are exponential only in the weighted depth and others are exponential only in channelwidth. This can be a significant improvement over the generic brute-force method, which is exponential in the total number of variables. Although the weighted depth is never smaller than the channelwidth, the weighted-depth algorithms have an advantage in that they use only linear space. In contrast, the channelwidth methods use space exponential in channelwidth.

We also introduce a concept of hierarchically specified formulas whereby large formulas are constructed in a recursive manner from copies of smaller formulas. This mimics the way larger circuits are constructed from smaller circuits and large graphs are sometimes constructed using copies of smaller graphs. We show that the value of such formulas can be found using generic algorithms whose complexity depends only on the size and structure of the description. The formula description can be exponentially smaller than the formula it describes.

Sometimes there is an (often nonunique) assignment that can be associated with a formula's value—for example, the assignment which minimizes a certain sum or which satisfies a conjunction of clauses. In these cases, generic enhanced algorithms will produce an assignment along with a value. These problems include all those problems known as "nonserial optimization problems" (NOPs) as well as constrained versions of these problems. We note that all {0,1}-linear programming problems are included among the constrained NOPs.

As the name suggests, the GSP applications also include problems centered around the set of assignments which satisfy a set of Boolean-valued terms. Not only is satisfiability itself modeled, but so are the other questions often asked about such problems. Some of these questions are how many satisfying assignments does the formula have, is the number of satisfying assignments even or odd, what is the maximum number of terms which can be simultaneously satisfied, and what is the maximum number of ones in a satisfying assignment? From the GSP viewpoint, these questions are nearly trivial variations on the same problem. The structure of each problem instance is the same regardless of the question asked and the only difference is whether the terms are thought of as mapping into TRUE and FALSE or into other values appropriate to the question asked. We note that among the problems modeled are complete problems in the complexity classes NP, CoNP, #P [34], D^P [23], OPT-P [16], MAX SNP [24], MAX Π_1 [22], PSPACE [17], and #PSPACE [4].

We have also found that the GSP model can be applied easily to many graph problems and that it clearly does not apply to certain other graph problems. Exactly which graph problems should be considered as GSPs is a nonmathematical question. There are graph problems whose semantics can only be described awkwardly and artificially as GSPs and it is a matter of taste which of these problems should be called GSPs.

A major advantage of the GSP framework is that it partitions the algorithmic considerations into four orthogonal issues: how to model the problem as a GSP, how to analyze the structure of problem instances, which generic algorithm to use, and how to implement the algebraic operations. Any of these four issues can be studied in isolation and the results then applied to the full spectrum of GSP problems.

The GSP model together with structure trees unify several general themes that have been widely used in the literature. These themes include "separator theorems" [20, 11, 33], "planarity" [18, 9, 26, 36], "graph treewidth" [1, 3, 5, 28, 2], and "bounded bandwidth" [21]. From a separator-theorem viewpoint, a structure tree can be interpreted as a method of displaying separator sets for a problem and its recursively defined subproblems. From the viewpoint of tree decompositions, the structure tree is an alternative way of displaying the same information. The GSP viewpoint "unifies" these themes in the sense that many of the techniques, especially insights into structure, can now be applied automatically to a much broader class of problems.

Our generic algorithms can be thought of as exhaustive methods modified to exploit subproblem independence. This is to be expected because the GSP model is a very general model and we are investigating methods that apply universally to all GSPs. When the generic algorithms are specialized to a particular class of formulas, it may be possible to enhance and fine tune the algorithms to exploit the semantics of the particular problem. For example, certain branches of the computation can sometimes be seen as unnecessary and thereby circumvented or pruned. Nevertheless, the gains just from subproblem independence are often very significant and sufficient to obtain many of the complexity bounds on structured problems found in the literature.

This paper is organized into 12 sections. In §2, we introduce the ideas of a formula, a formula value, and a GSP. In §3, we define structure trees and associated concepts. This section also presents the key connections between structure trees and subproblem independence, the connections which make the generic algorithms work.

In §4, we present the generic algorithms and bound the number of algebraic operations they perform.

In §5, we sketch the relationship between two parts of the structure tree, the α and β functions. These relationships suggest two approaches to finding good structure trees, namely, find a good α and construct β or find a good β and construct α. They also establish a small upper bound on the number of nodes needed for a good structure tree.

In §6, we discuss the relationships between weighted depth and channelwidth, the key parameters which determine the usefulness of a structure tree. In §7, we discuss the problem of finding good structure trees, even for instances where nothing is known about the structure in advance. The methods discussed are based on applying hypergraph techniques to a "formula hypergraph" representing the combinatorial structure of a formula.

In §8, we extend the ideas to hierarchically specified formulas. A fundamental property is proven that the hierarchical definition can, from a value standpoint, be equivalently regarded as defining an object or as defining a sequence of functions. The implications for efficient calculation of the value are presented.

In §9, we characterize the situations where a GSP has an optimal assignment in addition to a value and show how the generic algorithms can be modified to find the assignment along with the value without more than a constant multiplier on the complexity. The case of optimal assignments corresponds precisely to NOPs [30].

In §10, we discuss the concept of a "constrained GSP," in which the formula value is determined by a constrained set of formulas. We show that this concept can be reinterpreted back into the original model.

In §11, we discuss applications to Boolean satisfiability problems and their variations. We show that, from the GSP viewpoint, satisfiability, counting solutions, maximizing the number of terms satisfied, and several other variants all have the same structure and are, in fact, identical in appearance. Thus any way of solving one of these problems through good structure will apply to all these problems.

In §12, we discuss briefly the application of GSPs to graph problems. We discuss and illustrate the thesis that the key step in applying the theory to graph problems is to pick a representation of the problem as a GSP so that the GSP inherits the structure of the input graph.

Sections 9–12 demonstrate that the GSP model applies to a broad range of problems. However, the range is much broader than we have space to discuss. Additionally, it should be noted that the ideas behind subproblem independence and structure trees have natural extensions beyond GSPs [25].

The appendix gives the various algorithms discussed in the text. These algorithms incorporate major general-purpose themes in common use: top-down backtracking, top-down backtracking with table look-up or memoization, and bottom-up nonserial dynamic programming. The point of the appendix is that, in the GSP context, these themes can be coded generically using uninterpreted operators.

2. The GSP model.

Here we present a series of definitions leading to the central concept of a GSP. The GSPs are computational problems where the solution is a single value obtainable by considering all assignments to some set of finite-domain input variables. The value sought is an element of a commutative semiring (rather than just TRUE or FALSE), the semiring **times** operator is used instead of the Boolean **and** operator, and the semiring **plus** instead of the Boolean **or**.

DEFINITION 2.1. *A* commutative semiring *is specified by a* 5*-tuple* $(S, +, \cdot, 0, 1)$, *where* S *is a set containing elements* 0 *and* $1, +$ *is a commutative associative binary operator on* S *with identity* 0, $\cdot$ *is a commutative associative operator on* S *with identity* 1, $\cdot$ *distributes over* $+$ *(i.e.,* $a \cdot (b + c) = a \cdot b + a \cdot c$*), and* $1 \cdot 0 = 0$.

The identity elements are mathematically necessary so that sums and products are well defined for empty sets, namely $\sum_{a \in \phi} a = 0$ and $\prod_{a \in \phi} a = 1$. The condition $1 \cdot 0 = 0$ is really the distributive law for empty sums, namely $1 \cdot \sum_{b \in \phi} b = \sum_{b \in \phi} 1 \cdot b$.

Note that the Boolean lattice $(\{\text{FALSE}, \text{TRUE}\}, \vee, \wedge, \text{FALSE}, \text{TRUE})$ is a commutative semiring. We refer to this semiring as the *Boolean semiring.*

DEFINITION 2.2. *Given a set of variables* V, *an* assignment on V *is a pairing in which each variable* v *from* V *is paired with a value in the domain of* v. *The set of assignments to* V *will be designated by the notation* $\Gamma(V)$. $\Gamma(\emptyset)$ *contains one assignment, namely the empty set of pairs. For any assignment* γ, *we denote the variables in* γ *by* $VAR(\gamma)$ *(i.e.,* $VAR(\gamma) = V$ *if and only if* $\gamma \in \Gamma(V)$*). If* γ_1 *and* γ_2 *are assignments such that* $VAR(\gamma_1) \cap VAR(\gamma_2) = \emptyset$, *we let* $\gamma_1 + \gamma_2$ *be the assignment in* $\Gamma(VAR(\gamma_1) \cup VAR(\gamma_2))$ *formed by taking the union of the two assignments.*

Conjunction normal form (CNF) formulas can be described as a set of clauses. More generally, as in [31], a generalized CNF formula is described by a set of terms which evaluate to a Boolean value. For GSPs, we use terms which evaluate to an element of a commutative semiring. Continuing the analogy with Boolean formulas, we refer to these terms as "predicates." On occasion, we will need other kinds of "terms," including terms which behave like macros in hierarchical specifications.

DEFINITION 2.3. *A* base symbol *"f" is a symbol which has an associated integer* k *called the* arity *of "f" and as associated vector of* k *domains* $D_1, \ldots, D_k$, *where each* D_i *is a finite set. A* term *for "f" is a string of the form "*$f(x_1, \ldots, x_k)$*," where the* x_i *are variables or constants and the type of* x_i *is the domain* D_i. *If "f" is the name of a function which maps* $D_1 \times \cdots \times D_k$ *into a set* S, *any term for "f" is also called an* S-term. *If* S *is the set of elements from a semiring* R, *monoid* R, *or other algebraic object* R, *an* S*-term will also be called an* R-term. *If* R *is a commutative semiring, an* R*-term will also be called an* R-predicate *or simply a* predicate *if* R *is* *understood.*

For any term $P =$ "$f(x_1, \ldots, x_k)$," $|P|$ is defined to be $k+1$ and is called the size of P. $VAR(P)$ is defined to be the set of variables on the list $x_1, \ldots, x_k$. If γ is an assignment such that $VAR(\gamma) \supset VAR(p)$ and p is an S-term for some set S, we define $p[\gamma]$ to be the set element $f(d_1, \ldots, d_k)$, where d_i is the value assigned to variable x_i by γ. If P is a set of terms, define $VAR(P) = \cup_{p \in P} VAR(p)$.

In practice, we treat any expression as a term or predicate provided the meaning is clear. Thus we write "$x \vee z \vee \overline{y}$" instead of "$f(x, z, y)$," where symbol "$f$" denotes the function f defined by $f(v_1, v_2, v_3) = v_1 \vee v_2 \vee \overline{v_3}$.

DEFINITION 2.4. *A* formula *F is a pair (V, P), where V is a set of variables and P is a set of terms such that $V \supset VAR(P)$. If P is a set of S-terms for some set S, F is also called an* S-formula; *if S is the set of elements of a semiring R, semigroup R, or other algebraic object R, F is also called an* R-formula. *If R is a commutative semiring, the* value *of F, denoted by $VALUE(F)$ is defined by*

$$VALUE(F) = \sum_{\gamma \in \Gamma(V)} \prod_{p \in P} p[\gamma].$$

The size *of F, written $|F|$, is defined to be $|V| + \sum_{p \in P} |P|$.*

The pair (V, P) is a "formula" in the same sense that a set of clauses is a CNF formula. In the case of clauses, it is understood that the clauses are to be connected by the "and" operator. For R-formulas, we understand that the terms are to be connected by the "$\cdot$" operator (i.e., think of $\prod_{p \in P} p$ as the full formula).

So far, we have placed no restrictions on the formulas to be considered. Even so, any R-formula has a value and, as we shall see, there are many generic insights that can be derived about finding values, even if the domains, functions, and semirings are uninterpreted. Indeed, this paper is mainly about such results. However, to model specific computational problems, the formulas considered must come from some specified domain.

DEFINITION 2.5. *A* problem domain *$\mathcal{D}$ is a set of R-formulas for some commutative semiring R. A formula in $\mathcal{D}$ is called a* problem instance. *The* GSP *for $\mathcal{D}$ is to take formulas F from $\mathcal{D}$ as input and produce $VALUE(F)$ as output.* □

CNF satisfiability is modeled by the GSP for the domain consisting of formulas constructed from Boolean predicates that can be described by clauses. It is easily verified that, for this domain, the value of a formula is TRUE if and only if the formula has a satisfying assignment.

Other questions about CNF formulas can be modeled simply by reinterpreting the functions defined by the clauses. To model the problem of counting the number of satisfying assignments, take the semiring of nonnegative integers $(N, +, \cdot, 0, 1)$ and let a clause have the integer value 0 when an assignment makes it false and integer value 1 when an assignment makes it true. A product is 1 when an assignment satisfies all clauses and 0 otherwise and so the sum of products equals the number of assignments.

Notice how V is a nontrivial consideration when counting solutions. For example, if $P = \{$"$\overline{x} \vee y$","$x \vee \overline{z}$"$\}$, then $F_1 = (\{x, y, z\}, P)$ has four solutions, whereas $F_2 = (\{w, x, y, z\}, P)$ has eight solutions.

Most of what we say about formulas has nothing to do with problem domains. For example, we can analyze a formula's structure and find its value without regard to which problem domain it was taken from. Every formula, in fact, belongs to many problem domains. Thus we often make reference to GSPs and formulas without mention of a problem domain.

One way to solve a GSP is to use Algorithm 1, given in the appendix. This program has an inner loop which computes products for individual assignments and an outer loop which sums these products. It is best described as the "brute-force method." The program is "generic" in the sense that it works independently of any particular semiring or problem domain. We can understand the complexity of this algorithm by counting operations. If each variable has a domain of size D, there are $|P| \cdot D^{|V|}$ "multiplications," $D^{|V|}$ "additions," and $D^{|V|}$ evaluations of each predicate. Assuming a unit cost for each "+" and "$\cdot$" operation and a cost k for evaluating k-ary functions, the above counts imply a $\Theta(|F| \cdot D^{|V|})$ time complexity.

When the semiring and symbol sets of the problem domain are both finite, the above cost assumptions are reasonable. Infinite semirings or symbol sets are not necessarily a problem since the set of values which arise while solving a given problem instance is finite—for example, if we are trying to count the number of solutions, the number of bits involved is at most $|F|$. In any case, we will be content to count operations. Issues concerning the representation of semirings, the complexity of performing semiring operations, or the complexity of evaluating functions are mostly outside the scope of this paper.

Although we require that all variable domains be finite, the number of the domains need not be finite. In particular, it is sometimes useful to allow formula variables to take on $|V|$ values. In such cases, we encounter running times such as $\Theta(|F| \cdot |V|^{|V|})$ or, equivalently, $\Theta(|F| \cdot 2^{|V| \cdot \log V})$.

3. Structure trees. Given a formula $F = (V, P)$, we would like to compute $VALUE(F)$ faster than by the exhaustive method of Algorithm 1. We do not know how to do this in general but we can do it better if we can associate the formula with a suitable "structure tree" as defined below. The structure tree can be thought of as an organization of the formula which displays subproblem independence. This independence can be exploited in several ways, as given in §4, to compute the value. In this section, we present the structure tree concept and certain algebraic equations which explain the correctness of the faster algorithms given in §4. When we use the term "ancestor" and "descendant," we consider any tree node to be an ancestor and a descendant of itself.

DEFINITION 3.1. *Given a formula $F = (V, P)$, define a* structure tree S for F *to be a triple (T, α, β), where*

1. *T is a rooted tree with node set N,*
2. *$\alpha : V \longrightarrow N$ gives the* variable association,
3. *$\beta : P \longrightarrow N$ gives the* predicate association,
4. *for all y in V and p in P, if $y \in VAR(p)$, then $\alpha(y)$ is an ancestor of $\beta(p)$ in T.*

There are several concepts pertaining to the nodes of the structure tree that are needed later. We present these in the next definition.

DEFINITION 3.2. *Let $F = (V, P)$ be a formula and $S = (T, \alpha, \beta)$ be a structure tree for F with node set N. For each node n, define the following:*

1. *$A(n) = \{y \in V \mid \alpha(y) = n\}$, the variables* associated *with n.*
2. *$B(n) = \{p \in P \mid \beta(p) = n\}$, the predicates* associated *with n.*
3. *$AD(n)$ is the union of the $A(n')$ such that n' is a descendant of n.*
4. *$BD(n)$ is the union of the $B(n')$ such that n' is a descendent of n.*
5. *y in V is a* branch variable *at node n if and only if $\alpha(y)$ is an ancestor of n. Let $BV(n)$ be be the set of branch variables at n.*
6. *y in V is a* channel variable *at node n if and only if y is a branch variable*

and either (i) $n = \alpha(y)$ *or* (ii) *there is a* p *in* P *such that* y *is in* $VAR(p)$ *and* $\beta(p)$ *is a descendant of* n. *Let* $CV(n)$ *be the set of channel variables at* n.

In (6), case (i) is implied by case (ii) whenever y appears in some predicate. Case (i) insures that every variable is a channel variable of at least one node, an assumption necessary for certain proofs.

Intuitively, the nodes $\alpha(y)$ and $\beta(p)$, where y is in $VAR(p)$, delineate the "scope" of variable y and $CV(n)$ is the set of variables whose values must be "channeled" through n to connect $\alpha(y)$ and $\beta(p)$.

Example. Figure 1 shows a structure tree for formula $(\{w, x, w, z\}, \{$"$f(x, y)$," "$f(x, z)$," "$f(x, w)$," "$f(v, w)$"$\})$. The figure shows the sets $A(n)$, $B(n)$, $BV(n)$, and $CV(n)$ for each node n. The functions α and β can be inferred from A and B (e.g., $\alpha(x) = b$ because $x \in A(b)$).

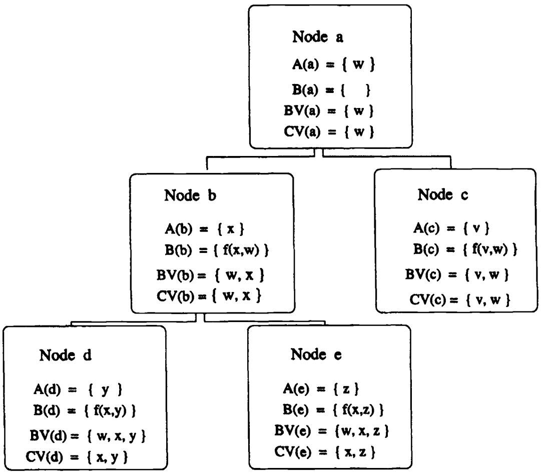


FIG. 1. *Structure tree for* $(\{w, x, y, z\}, \{f(x, w), f(x, y), f(x, z), f(v, w)\})$.

The algorithms described in the next section take a structure tree as part of their input and the complexities of these algorithms are characterized in terms of the parameters "weighted depth" and "channelwidth" defined as follows.

DEFINITION 3.3. *If* S *is a structure tree with node set* N, *we define* $WD(S)$, *the* weighted depth *of* S, *to be the maximum of* $\{|BV(n)| \mid for\, n \in N\}$. *We define* $CW(S)$ *to be the maximum of* $\{|CV(n)| \mid for\, n \in N\}$.

If F *is a formula, the minimum weighted depth over all structure trees for* F *is called the* weighted depth *of* F. *The minimum channelwidth over all structure trees for* F *is called the* channelwidth *of* F.

In the example, the weighted depth of (T, α, β) is 3 and the channelwidth is 2. There is another structure tree for F (with $\alpha(x)$ at the root) which also has weighted depth 3, but no structure tree has a smaller weighted depth. Therefore the weighted depth of the formula is 3. The channelwidth can never be less than the number of variables in a predicate so we know at once that channelwidth 2 cannot be improved upon.

Certain relationships among the concepts introduced above are needed to prove the correctness of various formulas and procedures given later. They are given by the following.

LEMMA 3.4. *Let* $F = (V, P)$ *be a formula,* $S = (T, \alpha, \beta)$ *a structure tree for* F, *and n a node of T. Then the following hold:*

1. $VAR(BD(n)) \subset CV(n) \cup AD(n)$,
2. $AD(n) \cap VAR(B(n)) \subset A(n)$, *and*
3. *for all children n' of n, $AD(n) \cap VAR(BD(n')) \subset A(n) \cup AD(n')$.*

Proof. These are all consequences of Definition 3.1(4). To prove (1), let x and p be such that $p \in BD(n)$ and $x \in VAR(p)$. Since $\alpha(x)$ must be above $\beta(p)$, $\alpha(x)$ is either above n or below n. In the case that $\alpha(x)$ is above, x will be in $CV(n)$ by definition. In the case that $\alpha(x)$ is below, x will be in $AD(n)$ by definition.

To prove (2), let x and p be such that $p \in B(n)$ and $x \in VAR(p)$. By definition, $\beta(p) = n$ so $\alpha(x)$ must be at or above n. If $\alpha(x) \in AD(n)$, $\alpha(x)$ also must be at n or below. Therefore, $\alpha(x) = n$.

To prove (3), let x and p be such that $p \in BD(n')$ and $x \in VAR(p)$. If x is also in $AD(n)$, then $\alpha(x)$ is in the subtree of T with root n and $\alpha(x)$ must be on the branch from n to $\beta(n)$, the branch going through n'. This implies that $\alpha(x)$ is at the root itself (i.e., $x \in A(n)$) or in the subtree with root n' (i.e., $x \in AD(n')$). □

So far, the concepts in this section are combinatorial concepts having nothing to do with the interpretation of the terms. That is, they depend only on memberships in the sets $VAR(p)$. This means that the structure of a formula, namely its structure trees, weighted depth, channelwidth, etc., is independent of the semantics of the formula. Hence the structural analysis of formulas (as in §7) can be carried out orthogonally to any consideration of the meaning of the formula.

Now we introduce concepts which require that the formulas be R-formulas for some semiring R. These concepts connect structure trees with formula evaluation.

DEFINITION 3.5. *Let $S = (T, \alpha, \beta)$ be a structure tree for formula $F = (V, P)$, let n be a node of T, and let γ be any assignment such that* (1) $VAR(\gamma) \cap AD(n) = \emptyset$ *and* (2) $VAR(\gamma) \cup AD(n) \supset VAR(BD(n))$. *Then let $E(n, \gamma)$ be the semiring member defined by*

$$E(n,\gamma) = \sum_{\gamma' \in \Gamma(AD(n))} \prod_{p \in BD(n)} p[\gamma + \gamma'].$$

The two conditions of the definition are necessary and sufficient for the expression to be well defined. Condition (1) is needed so that $VAR(\gamma') \cap VAR(\gamma) = \emptyset$, as required in Definition 2.2. Condition (2) is needed so that $VAR(\gamma + \gamma') \supset VAR(p)$ for all predicates p in the expression, as required in Definition 2.3. $VALUE(F)$ is identical to $E(r, \emptyset)$, where r is the root of the structure tree. $E(n, \gamma)$ can also be described as the value of subproblem $(AD(n), BD(n)[\gamma])$, where the notation $BD(n)[\gamma]$ represents the set of predicates obtained from $BD(n)$ by replacing variables from $VAR[\gamma]$ by their assigned values.

The formula in Definition 3.5 suggests an exhaustive method for computing $E(n, \gamma)$. The next result shows that $E(n, \gamma)$ can be computed from certain values associated with the children of n, values which can be viewed as solutions to subproblems. This key result is needed to prove the correctness of the generic algorithms in §4.

THEOREM 3.6. *If F, S, γ, and n are as defined in Definition* 3.5 *and node n has k children $n_1, \ldots, n_k$, then*

$$E(n,\gamma) = \sum_{\gamma' \in \Gamma(A(n))} \left(\prod_{p \in B(n)} p[\gamma + \gamma'] \right) \cdot \prod_{i=1}^{k} E(n_i, \gamma + \gamma').$$

Proof. We first prove that the $p[\gamma + \gamma']$ in the given expression are well defined. We know that $VAR(\gamma') = A(n) \subset AD(n)$ and so $VAR(\gamma') \cap VAR(\gamma) = \emptyset$ follows from condition (1) of Definition 3.5, and thus $\gamma + \gamma'$ is well defined. Because $B(n) \subset BD(n)$ and because of Lemma 3.4(2), condition (2) of Definition 3.5 implies $VAR(p) \subset VAR(\gamma + \gamma')$. Thus $p[\gamma + \gamma']$ is well defined.

Now consider if n_i and γ satisfy Definition 3.5. Because $AD(n_i) \subset AD(n)$ and $AD(n_i) \cap A(n) = \emptyset$, condition (1) for n_i and γ follows from condition (1) for n and γ. From Lemma 3.4(3), condition (2) for n and γ, and $BD(n_i) \subset BD(n)$, one can derive condition (2) for n' and γ.

Now that the right-hand side of the equation in the theorem has been shown to be well defined, we prove the equality by induction. If n has no children, $A(n) = AD(n)$, $B(n) = BD(n)$, and the equation is identical to Definition 3.5.

Finally, to prove the inductive step, substitute Definition 3.5 into both sides of the equation in the theorem. Applying the distributive and commutative laws to move products inside of sums, and using the two identities given below, the right-hand side can be transformed into the left-hand side. If V_1 and V_2 are disjoint sets of variables and P_1 and P_2 disjoint sets of predicates such that $V_i \supset VAR(P_i)$, the first identity is

$$\sum_{\gamma_1 \in \Gamma(V_1)} \sum_{\gamma_2 \in \Gamma(V_2)} \prod_{p \in P_1 \cup P_2} p[\gamma_1 + \gamma_2] = \sum_{\gamma \in \Gamma(V_1 \cup V_2)} \prod_{p \in P_1 \cup P_2} p[\gamma].$$

This identity holds because γ is in $\Gamma(V_1 \cup V_2)$ if and only if there are γ_1 in $\Gamma(V_1)$ and γ_2 in $\Gamma(V_2)$ such that $\gamma = \gamma_1 + \gamma_2$. For any γ_1 in $\Gamma(V_1)$ and γ_2 in $\Gamma(V_2)$, the second identity is

$$\left(\prod_{p \in P_1} p[\gamma_1]\right) \cdot \left(\prod_{p \in P_2} p[\gamma_2]\right) = \prod_{p \in P_1 \cup P_2} p[\gamma_1 + \gamma_2].$$

This identity holds because $p[\gamma_i] = p[\gamma_1 + \gamma_2]$ for all p in P_i, the value of p being independent of the values assigned to variables not in $VAR(p)$. □

Notice that the above proof makes use of each of the commutative, associative, and distributive properties of the binary operators. This explains why commutative semirings are the natural algebraic objects for the GSP model.

4. Algorithms based on structure trees. In this section, we discuss three algorithms (located in the appendix) for the GSP. Each algorithm requires that both a formula F and a structure tree for F be given as input. All three are "generic" in that they apply to all GSPs regardless of the problem domains. There are two main conclusions from this section. One is that structure trees are useful for organizing calculations for three general approaches, namely backtracking, backtracking with table look-up, and dynamic programming. The second is that weighted depth and channelwidth are the critical parameters in bounding the number of operations these algorithms perform.

The first algorithm is Algorithm 2. We refer to this algorithm as "generalized backtracking" since it evaluates particular partial assignments by considering all extensions and then "backs up" to try the next partial assignment. The key facts about this algorithm are as follows.

LEMMA 4.1. *When computation is initiated by calling $EVALUATE(r)$ from Algorithm 2, where r is the root of a structure tree for formula F, then the following hold for each node n:*

1. *When EVALUATE(n) is called, variables from $BV(n) - A(n)$ have an assignment γ_0.*
2. *EVALUATE(n) returns $E(n, \gamma_0)$.*
3. *EVALUATE(n) is called $|\Gamma(BV(n) - A(n))|$ times.*
4. *Summed over all calls, statements in the outer loop are executed $|\Gamma(BV(n))|$ times.*
5. *EVALUATE(r) returns $VALUE(F)$.*

Proof. Statement (1) is proved by induction starting from root r. $BV(r) = A(r)$ by definition so $BV(n) - A(n)$ is empty and trivially has an assignment. The assignment to $BV(n) - A(n)$ is extended by the "FOR statement" to an assignment to $BV(n)$ before the procedure is applied to child n' of n. But $BV(n)$ is $BV(n') - A(n')$ (this is immediate from Definition 3.2) and so (1) holds.

Statement (2) is now immediate because Algorithm 2 is a straightforward implementation of the formula in Theorem 3.6.

Whenever statement (3) is true, statement (4) must also be true since the outer loop is executed $|\Gamma(A(n))|$ time per call and multiplying the number of calls as given in statement (3) by $|\Gamma(A(n))|$ gives statement (4).

Statement (3) can now be proven by induction. It is true of the root r since *EVALUATE*(r) is called once and $BV(r) - A(r)$, being empty, has exactly one assignment. Now assume that statements (3) and hence (4) are true for node n and consider any child n' of n. *EVALUATE*(n') is called only from *EVALUATE*(n) and is done once each time the procedure goes through its outer loop. By statement (4), this happens $|\Gamma(BV(n))|$ times and $BV(n) = BV(n') - A(n')$. Thus statement (3) holds for n'.

Statement (5) is immediate since $E(r, \emptyset)$ is $VALUE(F)$ by Definition 3.5. □

From Lemma 4.1, we can bound the number of operations used to solve a GSP instance as follows.

THEOREM 4.2. *Let $F = (V, P)$ be a formula where each variable in V takes on at most D values. Let T be a structure tree for F having m nodes and weighted depth WD. If the procedure of Algorithm 2 is used with F and T, then*

1. *the number of "$\cdot$" operations is at most $(m + |P|) \cdot D^{WD}$;*
2. *the number of "+" operations is at most $m \cdot D^{WD}$;*
3. *each p in P is evaluated at most D^{WD} times.*

Proof. We consider part (3) first. A given predicate p is evaluated only when *EVALUATE* is called at node $\beta(p)$. The evaluation of p is done in the outer loop and so is done $|\Gamma(BV(n))|$ times by Lemma 4.1(4). This quantity is no greater than D^{WD}, and part (3) is proven.

The "$\cdot$" operation is performed once for each predicate evaluation and once for every procedure call (except the original call). From part (3), there are at most $|P| \cdot D^{WD}$ predicate evaluations and from Lemma 4.1(3), there are at most $m \cdot D^{WD}$ procedure calls.

The "+" operation is done once each time through the outer loop. By Lemma 4.1(4), this is at most D^{WD} per node, and hence part (2) is proven. □

In the next section, it is shown that, without changing WD, the structure tree T can be assumed to satisfy $m \leq 2 \cdot |V|$ and $m \leq 2 \cdot |P|$. Thus the bounds in parts (1) and (2) can be described as $3 \cdot |P| \cdot D^{WD}$ and $2 \cdot |V|^{WD}$ or $2 \cdot |P|^{\cdot WD}$. Under the unit-cost assumptions in §2, the time of the algorithm is $\Theta(|F| \cdot D^{WD})$, the largest contribution being the cost of evaluating predicates at k units per single k-ary function evaluation.

The space requirement is minimal. In addition to the space needed to store the structure tree, the procedure has a global variable for each variable in $|V|$, where each such variable must store a corresponding domain value, and two local variables x and y which must store any semiring elements generated by the procedure. Actually, only $O(WD)$ space is needed for variables since variables from different branches can share locations. We next consider Algorithm 3, the second of the three algorithms discussed in this section. It works similarly to Algorithm 2 except that tables are kept to remember the results of each procedure call and the tables are consulted at the beginning of any call to determine if the result the call can be read from a table rather than be recomputed. The key facts about this procedure are as follows.

LEMMA 4.3. *When computation is initiated by calling $EVALUATE(r)$ from Algorithm* 3, *where r is the root of a structure tree for formula F, then the following hold for each node n:*

1. *When $EVALUATE(n)$ is called, variables from $CV(n) - A(n)$ have an assignment γ_0;*
2. *$EVALUATE(n)$ returns $E(n, \gamma_0)$;*
3. *If n_0 is the parent of node n, then $EVALUATE(n)$ is called at most $|CV(n_0)|$ times;*
4. *$EVALUATE(r)$ returns $VALUE(F)$.*

Proof. To prove statement (1), observe that variables in $BV(n) - A(n)$ have assignments for the same reason as in Lemma 4.1(1). The result here is then immediate because $BV(n) \supset CV(n)$

To prove statement (2), first observe that γ_0 does meet the conditions imposed by Definition 3.5 because Definition 3.5(2) is implied by Lemma 3.4(1). Then observe that Algorithm 3 is a straightforward implementation of the formula in Theorem 3.6.

To prove statement (3), observe that $EVALUATE(n)$ is only called from inside $EVALUATE(n_0)$, and is only invoked at n for those calls at n_0 in which an assignment to $CV(n_0) - A(n_0)$ is seen for the first time. Those calls at n_0 which do invoke $EVALUATE(n)$ do so $|\Gamma(A(n_0))|$ times, once for each time through the variables loop. Multiplying the number of times the variables loop is used by the number of loop iterations gives $|\Gamma(CV(n_0))|$.

Statement (4) is immediate from statement (2) since $E(r, \emptyset)$ is $VALUE(F)$ by Definition 3.5. □

From Lemma 4.3, we can bound the number of operations used to solve a GSP instance via Algorithm 3.

THEOREM 4.4. *Let $F = (V, P)$ be a formula where each variable in V takes on at most D values. Let T be a structure tree for F having m nodes and channelwidth CW. If Algorithm* 3 *is used with F and T, then*

1. *the variable number of "$\cdot$" operations is at most $(m + |P|) \cdot D^{CW}$;*
2. *the number of "$+$" operations is at most $m \cdot D^{CW}$;*
3. *each p in P is evaluated at most D^{CW} times.*

Proof. This follows from Lemma 4.3 in the same way Theorem 4.2 follows from Lemma 4.1. □

Under the cost assumptions of the previous discussion, the time of the algorithm is $\Theta(|F| \cdot D^{CW})$. This is usually an improvement over the first algorithm since CW is never larger than WD. However, the space requirement now includes space fom m tables whose size could be as large as D^{CW}.

The third algorithm considered in this section is dynamic programming (Algorithm 4). The idea behind this algorithm is to work bottom-up in the structure tree, computing each table T_n (as defined above and in Algorithm 3) sometime after the tables for the children of n have been constructed. This algorithm also satisfies Theorem

4.4. We omit further analysis.

Backtracking with table look-up does hold a practical advantage over dynamic programming in that backtracking may leave some entries of the tables T_n uncomputed. On the other hand, dynamic programming can be organized to be more space efficient since tables can be discarded once the parent's table has been constructed. This is not a substantial difference from a complexity viewpoint.

5. Relationships between α and β. The relationship between α and β has implications for finding good structure trees and for bounding the number of nodes which must be considered by the generic algorithms. We omit proofs because they are easy and can be found in [32]. For ease of exposition, we assume that each variable of a formula appears in some predicate and that each predicate has at least one variable. Accommodating trivial variables and predicates is also discussed in [32].

Given a formula $F = (V, P)$, a rooted tree T, and a suitable α mapping V to tree nodes, a mapping β is easily constructed so that (T, α, β) is a structure tree for F. By "suitable," we mean that for each p in P there is a branch of T containing all $\alpha(y)$ for y in $VAR(p)$. This mapping β is defined by

$$\beta(p) = \text{ lowest node in } \{\alpha(x) \mid x \in VAR(p)\}.$$

In other words, $\beta(p)$ is placed as high in the tree as possible without violating Definition 3.1(4). This placement is optimal in that it gives the smallest channel variable sets. The pair (T, α) is called an *α-tree.* This concept opens an approach for finding structure trees, namely find a good α-tree and then construct the corresponding β. Given a formula $F = (V, P)$, an unrooted tree T, and an arbitrary β mapping P to tree nodes, a mapping α can be constructed, using any node as root, such that (T, α, β) is a structure tree for F. This α is defined by

$$\alpha(x) = \text{ lowest common ancestor of } \{\beta(p) \mid x \in VAR(p)\}.$$

In other words, $\alpha(x)$ is placed as low in the tree as possible without violating Definition 3.1(4). Again, this is optimal in that it gives the smallest branch-variable sets. The pair (T, β) is called a *β-tree.* This opens a second approach for finding structure trees, namely find a good β-tree and then construct the corresponding α. The mapping α constructed above has the property that a variable x is a channel variable at node n if and only if there are predicates p_1 and p_2 in P such that x is in $VAR(p_1) \cap VAR(p_2)$ and n is on a simple path between $\beta(p_1)$ and $\beta(p_2)$. This implies that channel variables can be defined by β alone and, therefore, channelwidth can be defined and computed directly from β just like weighted depth can be defined and computed directly from α.

Using the above ideas, one can modify a structure tree (in polynomial time) by moving various $\alpha(x)$ down the tree, moving various $\beta(p)$ up the tree, and deleting nodes n with only one child where $A(n)$ and $B(n)$ are empty. The result is a "compact" structure tree defined as follows.

DEFINITION 5.1. *Let $F = (V, P)$ be a formula in which every predicate has a variable and every variable belongs to some predicate. Let $S = (T, \alpha, \beta)$ be a structure tree for F. The structure tree is called* compact *if and only if the following conditions hold:*

1. *for v in V, $\alpha(v)$ is the lowest common ancestor of $\{\beta(p) \mid v \in VAR(p)\}$;*
2. *for p in P, $\beta(p)$ is the lowest node in $\{\alpha(v) \mid v \in VAR(p)\}$;*
3. *for nodes n in T, $A(n) = \emptyset$ or $B(n) = \emptyset$ implies n has at least two children.* □

The compact structure tree always has fewer nodes, smaller branch-variable sets, and smaller channel-variable sets than the original. Thus all the generic algorithms given here run more efficiently on compact trees. Compactness bounds the number of nodes as follows.

PROPOSITION 5.2. *If $S = (T, \alpha, \beta)$ is a compact structure tree for formula $F = (V, P)$ and N is the set of nodes of T, then $|N| \leq 2 \cdot |V|$ and $|N| \leq 2 \cdot |P|$.*

6. Relationship between weighted depth and channelwidth. There is a tradeoff between the algorithm based on weighted depth (Algorithm 2) and the algorithms based on channelwidth (Algorithms 3 and 4). The weighted-depth algorithm runs in linear space whereas the others use space exponential in channelwidth. However, the number of operations performed by the weighted-depth algorithm is exponential in the weighted depth whereas the others are exponential only in the channelwidth, which might be significantly smaller. Thus the relationship between weighted depth and channelwidth is of considerable interest.

Here we summarize the relationship without proof. By definition, a channel variable is also a branch variable so that, in any tree, the channelwidth is never more than the weighted depth. Also, it is easy to construct a formula and structure tree such that the structure tree has channelwidth 2 and arbitrary weighted depth. However, given a structure tree of good channelwidth, a tree of reasonably good weighted depth can be constructed easily as stated in the next theorem.

THEOREM 6.1. *Given a formula $F = (V, P)$ and a structure tree $S = (T, \alpha, \beta)$ for F with channelwidth CW, a structure tree $S' = (T', \alpha', \beta')$ for F can be constructed in polynomial time such that the weighted depth of S' is at most $CW \cdot (\log_2(|N|) + 1)$, where N is the set of nodes of T and also of T'.*

The theorem can be proven with a restructuring algorithm to construct T' from T given in [32]. It first changes the root of the tree to be one of the "middle nodes." (By middle node we mean a node with minimum maximum distance to any other node.) It then recursively applies this idea to the subtrees represented by each child of the new root r and edges from r are changed so that the roots of the restructured subtrees become the new children of r. The result is a new tree having the same node set as the original. The function β from the original structure tree is retained and a new α is computed using the principles of §5. Restructuring can also be achieved with a straightforward generalization of an algorithm from [6] so that weighted depth is $O(CW \cdot \log(|N|))$ and simultaneously channelwidth is O(CW). The new tree in this case has more nodes than the old.

7. Finding good structure trees. The algorithms described in §4 assume that a structure tree for the input formula is already known. It is therefore of considerable importance to understand the complexity of finding good structure trees since we would like to apply these techniques to situations where the structure tree is not given in advance. This includes situations where other structural information is known in advance.

As pointed out in §3, the concept of the structure tree for a formula is independent of the formula and depends only on a knowledge of which variables belong to which predicates. This information can be represented by a "formula hypergraph," defined as follows.

DEFINITION 7.1. *If $F = (V, P)$ is a formula, the* formula hypergraph *$HG(F)$ for F is the hypergraph (N, E), where N has one node n_v for each variable v in V, E has one edge e_p for each predicate p in P, and $n_v \in e_p$ if and only if $v \in VAR(p)$.*

The formula hypergraph provides links between the structure of formulas and previously studied structural concepts from graph and hypergraph theory. Some of these connections are as follow.

THEOREM 7.2. *Let $\mathcal{D}$ be a problem domain and let $\mathcal{H} = \{HG(F) \mid F \in \mathcal{D}\}$ be the set of hypergraphs for formulas in $\mathcal{D}$. Then the following hold:*

1. *If the hypergraphs in $\mathcal{H}$ have treewidth at most k, the formulas in $\mathcal{D}$ have channelwidth at most $k+1$.*
2. *If the hypergraphs in $\mathcal{H}$ are planar and each edge has at most m nodes, then each formula F in $\mathcal{D}$ has $O(\sqrt{|F|})$ weighted depth.*
3. *If the hypergraphs in $\mathcal{H}$ have bandwidth at most d and each edge has at most m nodes, then formulas in $\mathcal{D}$ have channelwidth at most $d \cdot m$.*
4. *If the hypergraphs in $\mathcal{H}$ satisfy a n^r recursive separator theorem* [19] *for $0 < r < 1$ and each edge has at most m nodes, then each formula F in $\mathcal{H}$ has weighted depth $O(n^r)$.*

Proof. Since we are not providing the definitions which appear elsewhere in the literature, we do not provide a proof. However, the results are almost immediate from the definitions. □

Theorem 7.2 together with the general theory make many insights available just by showing a problem can be modeled as a GSP. For example, it can be immediately confirmed that the planar version of such a problem can be solved in $2^{O(\sqrt{n})}$ operations and linear space (plus the polynomial time needed to find a structure tree of weighted depth $O(\sqrt{n})$ using planar separators).

We now discuss the problem of finding structure trees in the general situation. The methods discussed all start with the idea that the structure tree concept can be applied directly to hypergraphs using α to map nodes and β to map hyperedges and condition 4 of Definition 3.1 becoming "node v in edge p implies $\alpha(v)$ is above $\beta(e)$." The methods all find a structure tree for $HG(V, P)$ and then return the corresponding formula structure tree.

If $F = (V, P)$ is a formula and $V_0 \subset V$ and $P_0 \subset P$, we use the notation $[V_0, P_0]$ to represent the hypergraph with node set V_0 and edge set $\{VAR(p) \cap V_0 \mid p \in P_0\}$. Observe that $HG(V, P) = [V, P]$. It is easy to verify that, for any node n of a structure tree for $[V, P]$, the subtree with root n is a structure tree for $[AD(n), BD(n)]$. This enables a recursive approach to finding structure trees, namely the following:

1. Pick a set $V_0 \subset V$.
2. Construct a root node r.
3. Set $A(r) = V_0$ and $B(r) = \{p \in P \mid VAR(p) \subset V_0\}$.
4. Find the connected components of $[V - A(r), P - B(r)]$.
5. Construct a structure tree for each component and attach that tree to r.

The preferred choice of V_0 are sets which cause the graph to "separate" in step 4 into two or more connected components of substantial size. Under the right circumstances, a good choice of V_0 can be found by a "separator theorem" as in [20] or [11] and a structure tree built from recursive applications of the separator theorem. However, separator theorems do not have enough generality to explain all the circumstances where good separator trees arise. For example, recursive $O(n^r)$ separator theorems hold only for classes of graphs where the number of edges for graphs in the class is linear in the number of nodes (see Theorem 12 of [19]).

In [32], we incorporate exhaustive trials of all possible V_0 into the recursive approach. The result is an algorithm which takes graph (V, P) and integer d as input and, in $O(|F| \cdot |V|^d)$ time and polynomial space, determines if the given graph has a structure tree of weighted depth d and finds such a structure tree if one exists. Because this time bound is somewhat close to the $O(|F| \cdot D^d)$ operations, which suffice to solve a problem if a structure tree of weighted depth d is given, there are some interesting implications about exploiting structure even if no structure tree is given and no separator theorem applies.

Consider the implications of the above for SAT. If we allow time $|F|$ to evaluate a CNF formula F, SAT is solved by exhaustive search in $O(|F| \cdot 2^{|V|})$ time. In contrast, the above remarks imply the following result.

THEOREM 7.3. *SAT can be solved in* $O(WD \cdot |F| \cdot 2^{WD \cdot \log |V|})$ *time.*

Proof. Given F, apply the algorithm cited above repeatedly for $d = 1$, $d = 2$, ... until a separator tree is found. This happens when $d = WD$. Then apply Algorithm 2. Because Boolean operations can be performed in constant time, Theorem 4.2 implies the result. □

This last theorem shows that merely the *existence* of a good weighted-depth structure tree is sufficient to solve a SAT instance quicker than trying all cases, even though we are not given the structure tree as part of the input. With adjustments on the time bound for the domains and semirings involved, this idea applies to all GSPs.

Structure trees for hypergraphs have essentially the same information content as tree decompositions for hypergraphs. We now briefly explain this connection for readers familiar with tree decompositions (defined for graphs in [1]). A structure tree becomes a tree decomposition simply by taking $X(n) = CV(n)$ and viewing the tree as undirected. In the other direction, pick $\beta(p)$ to be one of the nodes such that $VAR(p)$ is in $X(n)$ and convert the resulting β-tree into a structure tree by the method of §5.

Because of the tight coupling of structure trees and tree decompositions, the tree-decomposition literature is quite useful for understanding structure trees. The intent of most papers on treewidth is to study graphs with bounded treewidth. The motivation for bounded treewidth is to obtain polynomial algorithms. (Bounded treewidth implies bounded channelwidth, which by Theorem 4.4 implies polynomial time when semiring operations are easy.) However, most of the results we have seen from the bounded-treewidth literature do generalize easily to hypergraphs and, except for conclusions about polynomial time, do not depend on the treewidth being of fixed size. We have already mentioned a procedure from [6] in §6. From [1], we infer that the problem of deciding if a formula has channelwidth less than some k is NP-complete, even if each predicate has only two variables. The literature has a succession of papers with procedures for finding tree decompositions (if any) for some given k. These procedures generalize without much difficulty to hypergraphs. The papers include [1, 27–29] and most recently [7]. The algorithms in [1] and [28] take space exponential in k (no problem when k is bounded). The algorithms in [29] and [27] are approximate in that they only find the best treewidth within a factor of 4. The approximation algorithm in [27] takes $O(27^{CW} \cdot CW \cdot |F|)$ time and polynomial space. The algorithm in [7] is exact and linear for fixed treewidth. However, its time is exponential in the cube of the treewidth. A treewidth algorithm explicitly for hypergraphs is given in [32]. It finds a structure tree of channelwidth k (if any) in $O(|F| \cdot |V|^{k \cdot \log |P|})$ time and uses only polynomial space. The algorithms imply that statements analogous to Theorem 7.3 can be made about channelwidth.

8. Hierarchically specified formulas. In this section we look at how formulas can be specified hierarchically using a sequence of templates. These templates can be visualized as macros which can be used to expand a given formula into a much larger formula, a formula which can be exponentially larger than the original. The good news is that the complexity of finding the value of this large formula depends only on the size and structure of the templates.

DEFINITION 8.1. *A* template *is a 4-tuple* $(``f", V, V', P)$ *where*

1. *"f" is the* defined symbol,
2. *V is an ordered set of variables called* parameters,
3. *V' is a set of variables with $V' \cap V = \emptyset$ called* local variables,
4. *P is a set of predicates with $V \cup V' \supset VAR(P)$.*

DEFINITION 8.2. *If $t = ``f(x_1, \ldots, x_k)"$ is a term and $M = (``f", V, V', P)$ is a template defining "f" with $VAR(t) \cap V =$ and $|V| = k$, then the* expansion of t by M *is the formula $(VAR(t) \cup V', P')$, where the predicates in P' are the predicates obtained from P by replacing occurrences of variables from V with the corresponding x_i. The same formula with variables in V' renamed will also be called an* expansion of t by M.

We always assume that when several expansions are involved, local variables are renamed for individual expansions so that they are different from the variables from any other expansion by the same or any other template.

DEFINITION 8.3. *Let Φ be a set of base symbols. A* hierarchical specification H *using Φ is a sequence $M_1, \ldots, M_k$ of templates $(``f_i", V_i, V_i', P_i)$ and a* base formula (V, P), *where the terms in P_i are $\Phi \cup \{f_1, \ldots, f_{i-1}\}$-terms and the terms in P are $\Phi \cup \{f_1, \ldots, f_k\}$-terms. The* formula specified *by H is the formula obtained from (V, P) by repeatedly expanding $\{f_1, \ldots, f_k\}$-terms until only Φ-terms remain.*

We want to discuss how obtaining structure trees for templates can be used to construct a structure tree for the expanded formula. We first need a structure tree concept for templates.

DEFINITION 8.4. *If $M = (``f", V, V', P)$ is a template, a* structure tree *for M is a triple $S = (T, \alpha, \beta)$, where S is a structure tree for formula $(V \cup V', P)$ and $V \subset A(r)$, where r is the root of T.*

THEOREM 8.5. *Let $M_1, ..., M_l, F_0$ be a hierarchical specification, T_0 a structure tree for formula F_0, and T_i for $1 \leq i \leq l$ a structure tree for template M_i. Let WD_i be the weighted depth and CW_i be the channelwidth of T_i for $0 \leq i \leq l$. Let F be the formula specified by the specification. Formula F has a structure tree with weighted depth no greater than $\sum_{i=0}^{l} WD_i$ and channelwidth no greater than $max\{CW_i \mid 0 \leq i \leq l\}$.*

Proof. We first describe how to construct a structure tree for a formula obtained from just one expansion. This will imply how a tree can be constructed from a series of expansions.

Let $F_0 = (V_0, P_0)$ be a formula, $M_1 = (``f", V_1, V_1', P_1)$ be a template, and $t = ``f(x_1, \ldots, x_k)"$ a term in P_0. Let $S_0 = (T_0, \alpha_0, \beta_0)$ be a structure tree for F_0 and $S_1 = (T_1, \alpha_1, \beta_1)$ be a structure tree for M_1. Let F be the result of expanding F_0 by t. With this notation, we now describe how to construct a structure tree (T, α, β) for F from the trees for F_0 and M_1.

The tree T is the tree obtained from T_0 and T_1 by making the root of T_1 be a child of node $\beta_0(t)$ in T_0. The variables of F are by definition $V_0 \cup V_1'$ and we define $\alpha(v) = \alpha_0(v)$ for v in V_0 and $\alpha(v) = \alpha_1(v)$ for v in V_1'. For terms p in $P_0 - \{t\}$, define $\beta(p) = \beta_0(p)$. If p is a predicate of F obtained by substitution for the parameters of some p_1 in P_1, define $\beta(p) = \beta_1(p_1)$.

To verify that the constructed tree is a structure tree, we must check condition (4) of Definition 3.1. Consider now a term p from P and a variable v in $VAR(p)$. If v in V_0 and p in P_0, $\alpha(v) = \alpha_0(v)$ is above $\beta(p) = \beta_0(v)$ because $\alpha_0(v)$ is above $\beta_0(v)$. If v in V_1' and p is obtained from p_1 in P_1 by substitution, $\alpha(v) = \alpha_1(v)$ is above $\beta(p) = \beta_1(p_1)$. Finally, consider v in V_0 and p obtained from p_1 in P_1. Since v is not in $VAR(p_i)$, v must be x_i for some i. Thus $\alpha(v) = \alpha_0(v)$ is above $\beta_0(t)$ and $\beta_0(t)$ is above the root of T_1 and hence above $\beta_1(p)$, so again (4) holds.

It is easy to verify that for node n of T originally from T_0, the channel variables are the same in both S and S_1. It is also easy to verify that the channel variables of a node n from T_1 is the set obtained from the original set of channel variables by replacing arguments with the corresponding x_i. In either case, the set of channel variables at a node is no larger than the original set.

Now consider the structure tree for the full expanded formula obtained by repeated use of the above construction. Since the channel variable sets never increase, the channelwidth of the constructed tree is no larger than any of the parts. Regarding weighted depth, any branch of the constructed tree consists of segments of at most one branch from each T_i. Thus the weighted depth of the branch is no more than the sum of the individual weighted depths. □

When the Φ-terms in Definition 8.3 are R-terms for commutative semiring R, then the hierarchical specification H describes a GSP. The weighted depth and channelwidth of the formula are bounded by Theorem 8.5 and any of the methods of §4 can be applied. However, the m and $|P|$ in Theorems 4.2 and 4.4 may be exponential in the size of the specification. Thus the number of operations can be exponential is specification size, even for bounded channelwidth. We now develop an alternative where the number of operations is bounded by the size and channelwidth of the templates. The improvements come from viewing templates as defining predicate symbols.

DEFINITION 8.6. *If $M = (\text{“}f\text{”}, V, V', P)$ is a template where the terms in P are all R-terms for some commutative semiring R, then M is also called an* R-template *and the function “f” from $\Gamma(V)$ to R defined by*

$$f(\gamma) = \sum_{\gamma' \in \Gamma(V')} \prod_{p \in P} p[\gamma + \gamma']$$

is called the predicate function defined by M.

This definition enables us to view a hierarchical specification as a sequence of function definitions. The next result says that the value implied from the “function” view equals the value implied by the “macro” view.

THEOREM 8.7. *Let Φ be a set of function names for functions into commutative semiring R. Let $M_1, \ldots, M_k, F_0$ be a hierarchical specification and F the formula described by the specification. Then each M_i is an R-template and F_0 is an R-formula. Furthermore,*

$$VALUE(F_0) = VALUE(F).$$

Proof. The terms of M_1 are all Φ-terms and hence they are R-terms and M_1 thus defines a function to R. This implies that the terms of T_2 are all R-terms and so on by induction. Finally, the terms of F_0 are all R-terms because they are constructed from Φ and the function symbols defined by the templates. All that remains to be shown is the claim about values.

We first consider the case where there is just one template $T_1 = (\text{“}f\text{”}, V_1, V_1', P_1)$ and formula $F_0 = (V_0, P_0)$ has one f-term, namely $t = \text{“}f(x_1, \ldots, x_k).\text{”}$ Let $F = (V_0 \cup V_1', (P_0 - \{t\}) \cup P_1')$ be the expansion of f_0 by t, where P_1' is the set P_1 with the x_i replacing the corresponding v_i from V_1. We want to show $VALUE(F_0) = VALUE(F)$. By definition,

$$VALUE(F_0) = \sum_{\gamma_0 \in \Gamma(V_0)} \left(\prod_{p \in P_0 - \{t\}} P[\gamma_0] \right) \cdot t[\gamma_0].$$

Consider $t[\gamma_0]$, which is "$f(x_1,\ldots,x_k)$"$[\gamma_0]$. The list $x_1,\ldots,x_k$ and γ_0 imply a corresponding assignment γ_0' in $\Gamma(V_1)$ where each v_i in V_1 is assigned the value assigned to x_i by γ_0. The value of $t[\gamma_0]$ is defined to be

$$\sum_{\gamma'\in\Gamma(V_1')}\prod_{p_1\in P_1} p_1[\gamma'+\gamma_0'].$$

Now $p_1[\gamma'+\gamma_0'] = p[\gamma'+\gamma_0]$, where p is the result of replacing v_i in P_1 by the corresponding x_i. Thus $t[\gamma_0]$ can be written as

$$\sum_{\gamma'\in\Gamma(V_1')}\prod_{p_1\in P_1'} P[\gamma'+\gamma_0'].$$

Thus $VALUE(F_0)$ is

$$\sum_{\gamma_0\in\Gamma(V_0)}\left(\prod_{p\in P_0-\{t\}} P[\gamma_0]\right)\cdot\sum_{\gamma'\in\Gamma(V_1')}\prod_{p\in P_1'} P[\gamma'+\gamma_0].$$

Applying the distributive law gives

$$\sum_{\gamma_0\in\Gamma(V_0)}\sum_{\gamma'\in\Gamma(V_1')}\prod_{p\in(P_0-\{t\})\cup P_1'} p[\gamma'+\gamma_0],$$

which is $VALUE(F)$.

Having proven the result when F is the result of just one expansion, the result follows by induction. □

By treating a hierarchical specification as a sequence of function definitions, we can obtain alternative algorithms for computing the value. The most straightforward approach is to make a table for the predicate function defined by the first template M_1, then a table for the predicate defined by M_2, etc., and finally compute the value of the formula F_0. If the template values are computed with the backtracking of Algorithm 2, the space will be exponential in the arity of the templates and the number of operations will depend on the templates' weighted depths. If computed with Algorithms 3 or 4, the space will be exponential in template channelwidth which could be substantially larger, but the number of operations will be smaller depending only on channelwidth. More precisely, this case can be stated formally as follows.

THEOREM 8.8. *Let $H = (M_1,\ldots,M_k,F_0)$ be a hierarchical specification where each variable takes on at most D values. Let $(T_1,\ldots,T_k,T_0)$ be structure trees for the templates and formula of H. Let CW be the maximal channelwidth of any of the structure trees T_i. Then the value of H can be found algebraically where the number of "$\cdot$" operations, the number of "$+$" operations, and the number of predicate evaluations are bounded by $2\cdot|H|\cdot D^{CW}$.*

Proof. The method of computing the value has already been described. Because template variables are all channel variables of the root, the tables giving a value for each assignment to template parameters is the projection of the root table computed in Algorithms 3 or 4. By using compact structure trees, we can insure by Proposition 5.2 that the number of nodes in all the structure trees is no more than twice the number of specification variables. The result follows from Theorem 4.4. □

The importance of this result is that the bound contains $|H|$ instead of $|F|$, where F is the formula obtained by expanding H, a formula which can be exponentially larger than H. Finally, these observations hold for the analogous problems over any finite algebraic structure, e.g., the problems in [12].

9. Optimization problems. In this section, we consider the circumstances in which the GSP model can be used to answer questions of the following form: "given a formula (V, P), find an assignment γ_0 in $\Gamma(V)$ which results in the best value for $\prod_{p \in P} p[\gamma_0]$." Intuitively, such a γ_0 is properly referred to as an "optimal assignment." In this section, we argue that the GSP concept applies precisely to those optimization problems where the question can also be posed as a question of the form "find an assignment γ_0 in $\Gamma(V)$ which minimizes $\sum_{p \in P} p[\Gamma_0]$," and we present a generic algorithm for solving such problems. Since problems of this form are known as NOPs [30], we can say that the GSP models all NOPs.

If the value of a formula is to be an "optimal" value associated with an "optimal assignment," then the optimal assignment must (at least) satisfy the following definition.

DEFINITION 9.1. *Given an R-formula $F = (V, P)$ for commutative semiring R, an* optimal assignment *for F is an assignment γ_0 in $\Gamma(V)$ such that*

$$VALUE(F) = \prod_{p \in P} p[\gamma_0].$$

This concept of "optimal assignment" seems unrelated to any concept of "best value," but the next two propositions show that such an assignment and the value it produces are, in fact, the solution to a minimization problem.

PROPOSITION 9.2. *Let $R = (S, +, \cdot, 0, 1)$ be a commutative semiring. Then every R-formula has an optimal assignment if and only if, for all a and b in S,*

$$a + b = a \text{ or } a + b = b.$$

Proof. If the condition on R holds, then for any $a_1, \ldots, a_n$ in S, $\sum a_i = a_k$ for some k, $1 \leq k \leq n$. Since $VALUE(F)$ is a sum of assignments to $\prod_{p \in P} p$, the sum must be equal to some $\prod_{p \in P} p[\gamma_0]$ and γ_0 is an optimal assignment.

If, for some a and b, $a + b \neq a$ and $a + b \neq b$, define $f : \{0, 1\} \to S$ by $f(0) = a$ and $f(1) = b$. Let $F = (\{x\}, \{\text{"}f(x)\text{"}\})$. $VALUE(F) = a + b$ and neither of the two assignments $x = 0$ or $x = 1$ is an optimal assignment. □

It should be noted that all formulas in a problem domain might have a solution even if $a \neq a + b \neq b$ for some a and b. This is because the problem domain might not include the formula F from the proof. Nevertheless, the proposition indicates special attention should be given to the condition "$a + b = a$ or $a + b = b$." This condition takes us immediately to commutative ordered monoids and nonserial optimization.

PROPOSITION 9.3. *If $R = (S, +, \cdot, 0, 1)$ is a commutative semiring in which $a + b = a$ or $a + b = b$ for all a, b in S, then the binary relation $\leq$ on S defined by*

$$a \leq b \Leftrightarrow a + b = a$$

is a total order on S. Furthermore,

$$a + b = \min(a, b)$$

and

$$a \leq c \, and \, b \leq d \, implies \, a \cdot b \leq c \cdot d.$$

Proof. The first claim is easily verified. The last claim is slightly tricky. Suppose $a = a + c$ and $b = b + d$. The condition on $+$ implies $x + x = x$ for all $x \in s$. Thus

$ab = ab+ab = (a+c)(b+d)+ab = (ab+cb)+(ab+ad)+cd = (a+c)b+a(b+d)+cd = ab+cd$. □

Because plus is really a minimization operator, we see that the value of the formula is actually the minimum value of $\prod_{p\in P} p[\gamma]$ over all possible assignments $\gamma \in \Gamma[V]$ and the optimal assignment is the assignment which achieves this minimum. Hence the appropriateness of the adjective "optimal."

A further change in notation puts this problem into standard nonserial dynamic optimization notation. In addition to changing "+" to "min," change "·" to "+," "0" to "∞" (the identity element for min), and "1" to "0" (the identity element for +). Thus $(S,+,\cdot,0,1)$ becomes $(S,\min,\cdot,\infty,1)$. With these changes in notation,

$$VALUE((V,P)) = \min_{\gamma\in\Gamma(V)} \sum_{p\in P} p[\gamma]$$

and an optimal assignment is a γ_0 in $\Gamma(V)$ which achieves the minimum.

To argue the converse that nonserial optimization can be modeled by a GSP, we first define nonserial optimization in a very general way.

DEFINITION 9.4. *An* ordered commutative monoid *is specified by a* 5*-tuple* $G = (S,+,0,\leq,\infty)$, *where* $+$ *is a commutative associative binary relation on* S *with identity* 0 *and* $\leq$ *is a total order on* S *such that* $a \leq \infty$ *for* a *in* S *and* $a \leq c$ *and* $b \leq d$ *implies* $a+b \leq c+d$. *If* $\mathcal{D}$ *is a set of G-formulas, the* NOP *for* $\mathcal{D}$ *is to take a formula* $F = (V,P)$ *from* $\mathcal{D}$ *and find both the minimum value of the set*

$$\left\{ \sum_{p\in P} p[\gamma] \mid \gamma \in \Gamma(V) \right\}$$

and an assignment γ_0 *for which the minimum is obtained.*

Sometimes an ordered monoid is defined without an ∞. The integers and rationals under addition are such monoids. However, an ordered monoid without ∞ can always have an ∞ appended and + extended so that $a+\infty = \infty+a = \infty$ for all a in $S\cup\{\infty\}$.

Proposition 9.3 can now be interpreted as saying that any optimization problem based on GSPs can also be interpreted as a NOP. The semiring + is interpreted as min and the · as monoid +. The next result says that the converse is true.

THEOREM 9.5. *For any ordered commutative monoid* G, *there is a commutative semiring* R *such that any G-formula* F *is an R-formula, the minimum value of G-formula* F *is* $VALUE(F)$, *and the optimal assignments for the G-formula are the same as the optimal assignments for the R-formula.*

Proof. Suppose $G = (S,+,0,\leq,\infty)$. Let R be $(S,\min,+,\infty,0)$. R is easily verified to be a commutative semiring. The distributive law, namely $a+\min(b,c) = \min(a+b,a+c)$, follows because $\min(b,c) = b$ implies $b \leq c$, which implies $a+b \leq a+c$, which implies $\min(a+b,a+c) = a+b$. Because G-terms map into S, they are also R-terms and hence G-formulas are S-formulas. The problem equivalence follows for reasons already discussed. □

We now consider how the generic algorithms can be modified to find an optimal assignment together with the value. Roughly speaking, the modifications are achieved by remembering the partial assignments which cause the best partial result to occur.

The modified version of Algorithm 2 is Algorithm 5. The modifications are designed so that when a call is completed, the global variables from $AD(n)$ have the assignment which (together with values assigned to $BV(n) - AD(n)$) gives the returned value. We have also switched from semiring notation in Algorithm 2 to ordered monoid notation. The performance of the algorithm is as follows.

THEOREM 9.6. *Let $F = (V, P)$ be a formula where each variable in V takes on at most D values. Let T be a structure tree for F having m nodes and weighted depth WD. If Algorithm 5 is applied to F and T, then*

1. *the number of "+" operations is at most $(m + |P|) \cdot D^{WD}$;*
2. *the number of* "min" *operations is at most $m \cdot D^{WD}$;*
3. *each p in P is evaluated at most D^{WD} times;*
4. *the number of "save a value" operations is at most $2 \cdot |V| \cdot m \cdot D^{WD}$.*

Proof. Except for (4), this is Theorem 4.2 translated to NOP notation. The saves only occur after a comparison or just prior to a return. Since up to $|V|$ values need to be saved, (2) implies (4). □

We note that Algorithm 5 becomes more efficient if we change "if $x \leq y$" to "if $x < y$." However, the programmer must then also take into account the possibility that the local variables may not contain any assignment if the value ∞ is returned. Other improvements to cut off unnecessary computations are possible but are beyond the scope of this presentation. The traditional method of solving NOPs is to use nonserial dynamic programming similar to Algorithm 4. However, our approach indicates that the full range of GSP techniques are available.

Because NOPs can be regarded as special GSPs, the methods of §8 define hierarchical NOP and tell us how to find their values efficiently. To print an optimal assignment could take time exponential in the size of the hierarchical specification simply because the number of variables may be exponential in this size. A reasonable alternative method of representing an assignment is to give an assignment of variables in the base formula together with a table for each template showing an optimal assignment of local variables for each assignment of parameter variables. From this information, the assignment to any particular variable in the expanded formula is easily computed. The appropriate tables can be computed by enhancing the algorithm of Theorem 8.8 with the techniques discussed earlier in this section.

10. Constrained GSPs.

The value of a GSP formula is, by definition, determined from a consideration of the set of all variable assignments. Here we consider situations where the value we want depends only on a subset of the assignments. We call these "constrained formulas" because the assignment subset is specified by a set of "constraints." However, instead of developing a separate theory for constrained GSPs, we show how constrained GSPs (CGSPs) are easily transformed back into ordinary unconstrained GSPs. This further illustrates the scope of the GSP model and makes the model easier to apply since it is often easy to see that a given problem can be posed as a CGSP.

DEFINITION 10.1. *If $R = (S, +, \cdot, 0, 1)$ is a commutative semiring and B is the Boolean semiring, then a* constrained R-formula *is given by a triple $C = (V, P, Q)$, where (V,P) is an R-formula and (V, Q) is a B-formula. The value of C, denoted by $VALUE(C)$, is defined by*

$$VALUE(C) = \sum_{\gamma \in \Gamma_0} \prod_{p \in P} p[\gamma],$$

where

$$\Gamma_0 = \{\gamma \in \Gamma(V) \mid q[\gamma] = \text{TRUE}\, \textit{for all}\, q \in Q\}.$$

If $\mathcal{D}$ is a set of constrained R-formulas, then the constrained GSP *(or CGSP) for $\mathcal{D}$ is to take a constrained formula from $\mathcal{D}$ and find its value.*

Note that if $\gamma_0 = \emptyset$, then the sum is on the empty set and $VALUE(C) = 0$.

The transformation from CGSP to GSP employs a very simple kind of term transformation achieved by mapping the term output into another set:

DEFINITION 10.2. *If r is a function from S_1 to S_2 and $p =$ "$f(x_1, \ldots, x_k)$" is an S_1-term, then we use the notation $r \circ p$ to mean an S_2-term equivalent to the expression "$r(f(x_1, \ldots, x_k))$". If Q_1 is a set of S_1-terms, we call the set $Q_2 = \{r \circ p \mid p \in Q_1\}$ a* reinterpretation *of Q_1* by *r and we write $Q_2 = r \circ Q_1$.*

THEOREM 10.3. *If (V, P, Q) is a constrained R-formula, Q has a reinterpretation $Q' = r \circ Q$ such that formula $(V, P \cup Q')$ is an R-formula having the same value.*

Proof. Let $R = (S, +, \cdot, 0, 1)$ be the commutative semiring for P. To get Q', use the reinterpretation r, where $r(\text{TRUE})$ is 1 and $r(\text{FALSE})$ is 0. Observe that $\sum_{t \in P \cup Q'} t[\gamma]$ is equal to $\sum_{p \in P} t[\gamma]$ whenever $q[\gamma] = 1$ for all q in Q' and is equal to 0 if $q[\gamma] = 0$ for some q in Q'. The result about values follows easily. □

Remark. Because Q and $r \circ Q$ have identical hypergraphs, Theorem 10.3 shows that it is the structure of P and Q combined (i.e., of $P \cup r \circ Q$) that must be considered in order to apply the generic algorithms to (V, P, Q). An important special case is the "linear case" where the predicates in P all have the form "$f_i(x_i)$." In this case, any structure tree (S, α, β) for (V, Q) is, for structural purposes, identical to a structure tree (S, α, β') for $(V, P \cup r \circ Q)$, where $\beta'(r \circ q) = \beta(q)$ and $\beta'(\text{"}f_i(x_i)\text{"}) = \alpha(x_i)$. The branch variables at each tree node are the same in both cases and so are the channel variables. Thus both cases have the same weighted depth and the same channelwidth.

By analogy, for an ordered commutative monoid G, we can define a "constrained G-formula" and hence a "constrained NOP" (CNOP).

COROLLARY 10.4. *If $G = (S, +, 0, \leq, \infty)$ is an ordered commutative monoid, any constrained G-formula F can be transformed into a GSP-formula F' such that*

1. *F and F' have the same value and same optimal assignments whenever the constraints are satisfiable;*
2. *the value of F' is ∞ and all assignments are optimal whenever the constraints are unsatisfiable.*

Remark. In the case where the G-terms of F have the form "$f_i(x_i)$", the structure of the problem is captured by the structure of the constraints for the same reason as in the previous remark. We call this the "linear objective function case."

We note that {0,1}-linear programs are a GSP with linear objective functions. The variable domains are $\{0, 1\}$, the linear constraints describe predicates, and the objective function terms are the single variable expressions $c_i v_i$.

11. Boolean satisfiability problems. By a "Boolean formula," we mean a B-formula where B is the Boolean semiring ($\{\text{TRUE}, \text{FALSE}\}$, $\vee$, $\wedge$, FALSE, TRUE). By a "Boolean satisfiability problem," we mean a GSP whose problem domain is some set of Boolean formulas. As observed in §2, the question "does a Boolean formula F have a satisfying assignment?" has answer "yes" if and only if $VALUE(F) = \text{TRUE}$. Thus SAT, 3SAT, and the problems from [31] are Boolean satisfiability problems. Satisfiability is one of several questions commonly asked questions.

DEFINITION 11.1. *We define the following problems for Boolean formulas F:*

- SATISFIABILITY. *Does F have a satisfying assignment?*
- COUNTING. *How many satisfying assignments does F have?*
- MAXIMIZATION. *What is the maximum number of predicates from F that can be simultaneously true?*
- PARITY. *Is the number of satisfying assignments odd or even?*
- UNIQUENESS. *Does F have a unique solution?*

TABLE 1
Semirings for Boolean formula questions.

PROBLEM	S	+	·	0	1
SATISFIABILITY	{T,F}	OR	AND	F	T
COUNTING	integers	+	·	0	1
MAXIMIZATION	naturals	max	+	0	0
PARITY	{0,1}	XOR	AND	0	1
UNIQUENESS	{0,1,2}	$\min(a+b,2)$	$\min(a\cdot b,2)$	0	1

Each of the defined problems can be expressed in a well-known way as a language-recognition problem, namely recognize the set of satisfiable formulas, recognize the set of pairs (F,k), where F has at least k satisfying assignments, etc. For CNF formulas, the corresponding language problems are known as SAT, #SAT, MAX SAT, PARITY SAT, and UNIQUE SAT and are complete for NP, #P [34], MAX-SNP [24], PARITY-P, and D^P [35]. The hierarchically specified versions of 3SAT and #3SAT are PSPACE- and #PSPACE-complete, respectively [13].

Recalling the concept of reinterpretation given in Definition 10.2, representations for each problem in Definition 11.1 are obtainable as follows.

PROPOSITION 11.2. *For each of the problems given in Definition* 11.1, *there is a commutative semiring* $(S,+,\cdot,0,1)$ (*given in Table* 1) *and a reinterpretation function* $r : \{\text{FALSE},\text{TRUE}\} \to S$ *such that the answer for Boolean formula* $F = (V,P)$ *is* $VALUE(V, r\circ P)$.

Proof. The semirings are given in Table 1 and, in each case, r maps FALSE to 0 and TRUE to 1. In the case of UNIQUENESS, think of "2" as "more than one." In this case, $VALUE(V,P)$ returns three possible answers, namely "no satisfying assignment," "one (unique) assignment," or "more than one assignment." For each problem, the assertion about $VALUE(V, r\circ P)$ is easy to verify. □

Because formulas (V,P) and $(V, r\circ P)$ have the same structure (i.e., they have identical formula hypergraphs), the problems from Definition 11.1 have equal complexity with regard to finding good structure trees and with regard to the number of operations performed by the generic algorithms. There may, however, be cost differences associated with performing the operations. For example, COUNTING can involve numbers with $|V|$ bits whereas SATISFIABILITY involves only Boolean values. In this same regard, care must be taken when extrapolating to a hierarchical specification H. For example, COUNTING then can involve numbers with $2^{|H|}$ bits.

Of the five problems from Definition 11.1, SATISFIABILITY and MAXIMIZATION are optimization problems, and hence optimal assignments can be computed along with the value. Although UNIQUENESS is not an optimization problem ($1+1 \neq 1$ contrary to Proposition 9.2), the uniquely satisfying assignment (if any) can be found by very similar methods.

Closely related to the above are problems of the form "find a satisfying assignment which minimizes $\sum_{x_i\in V} f_i(x_i)$." One of these problems is "find the satisfying assignment with the largest number of variables assigned TRUE," which is described by setting $f_i(\text{FALSE}) = 0$ and $f_i(\text{TRUE}) = 1$ for all i. This problem is MAXΠ_1-complete [22]. Another is "find the lexicographically smallest satisfying assignment," which is described by setting $f_i(F) = 0$ and $f_i(T) = 2^{|V|-i}$ for all i. This last problem is OPT-P-complete [16]. Both problems are constrained NOPs with a linear objective function and, as discussed in the remarks of §10, their structure is essentially just the structure of the constraints inherited from the original Boolean formula.

One implication of the fact that all problems in this section have identical structure is that Theorem 7.3 applies to all these problems, not just for SAT, after possible adjustments for the complexity of performing the corresponding semiring operations.

12. Graph problems. A fundamental question about graphs is the following: for which hard graph problems are highly structured input graphs easier to solve than the general case? A sufficient condition is provided by the GSP model: structured graphs are easier to solve when the semantics of the graph problem can be described by a GSP in such a way that the formula hypergraph inherits the structure of the graph. This inheritance is most direct and useful when the formula hypergraph and input (hyper)graph are identical. This happens if the GSP has variables corresponding to each node, predicates corresponding to each hyperedge, and each predicate has the variables which correspond to its hyperedge nodes. We call this the node-variable edge-predicate (NVEP) case.

A good example of NVEP is graph k coloring. To model this as a GSP, introduce a variable for each graph node and let the variables range over a set of k colors. For each graph edge (n_1, n_2), introduce a predicate "$f(v_1, v_2)$", where v_1 and v_2 are the variables corresponding to n_1 and n_2 and $f(v_1, v_2)$ is TRUE if and only if $v_1 \neq v_2$. Then the graph has a k coloring if and only if the value of the GSP is TRUE. This is an example of a Boolean satisfiability problem as discussed in the previous section.

The structure of a graph is also passed directly to a GSP when the problem semantics are described by NVEP constraints and a linear objective function to be maximized or minimized. (See the Remarks in §10.) A good example of this is minimum node cover for hypergraphs. There is an obvious representation where the node variables take on the value TRUE (for nodes included in cover) or FALSE (for nodes not included), there is a constraint $v_1 \vee \cdots \vee v_k$ for each hyperedge $\{x_1, \ldots, x_k\}$, and there is an objective term "$f(v)$" for each variable, where $f(\text{TRUE}) = 1$ and $f(\text{FALSE}) = 0$.

A simple extension of the NVEP case is to allow each predicate to contain an additional variable which is associated with the corresponding hyperedge. With minor adjustments to the structure tree, this extension can be accommodated with an increase of at most one in weighted depth and treewidth/channelwidth. This is essentially the edge condition composition (ECC) case of [5]. We say "essentially" because ECC is defined only for ordinary graphs, the definitions include conditions on certain monoids and the complexity of monoid operations, and the results are oriented toward bounded-treewidth graphs. However, the theme of [5] is the same as in this section, namely that graph structure can be exploited to solve a graph problem more quickly *provided the semantics of the graph problem has a suitable semantics*, and the proofs of [5] can be viewed as showing that the many graph problems considered are GSPs.

Some problems are more naturally represented with variables for each edge and predicates for each node, a case we call EVNP. For example, minimal edge cover is most naturally represented by EVNP constraints and a linear objective function. Here the structure of the graph is passed on but not so directly. (The formula hypergraph is actually the hypergraph dual of the input.) A structure tree for the graph can be converted to a structure tree for the formula, but the weighted depth and channelwidth can increase by a factor of k, where k is the size of the largest hyperedge. (k=2 for ordinary graphs.) This case relates to the local condition composition (LCC) case of [5] in the same way that NVEP relates to ECC. We omit further details.

The minimum edge-cover problem just discussed can also be represented (less naturally) as a NVEP problem. Let the variable for each node range over the set of edges for that node. For each hyperedge e, construct an integer-valued predicate which is one if any of the arguments are equal to edge e and is zero otherwise. For any assignment, the set of edges taken on by the node variables is an edge cover and the number of predicates with value 1 is the size of this edge cover. Thus the value of the formula under semiring $(N, \min, +, \infty, 0)$ is the minimum cover.

The last example illustrates the use of "instance-specific domains," in this case variables which take on edge names as values. With the use of instance-specific domains, even HAMILTONIAN CIRCUIT and TRAVELING SALESMAN can be described as GSPs [32]. From a computational viewpoint, the value of such representations diminishes as the domains become more contrived. It should also be noted that there are nonalgebraic ways of exploiting structure trees that may be computationally more appropriate. Thus whether certain graph problems should be called GSPs is really a matter of taste.

Some graph problems are known to be hard even when the graphs have a very simple structure. One such problem is BANDWIDTH (problem [GT40] from [10]) which is NP-complete even for trees (trees have channelwidth 2 and treewidth 1).

When the semantics of a graph problem are described as a GSP, we have found that the hierarchical methods of §8 frequently apply. This is the subject of another paper.

Appendix.

ALGORITHM 1. *Brute force.*

Input: Formula (V, P)
Variables: v in V, x for accumulating a sum, y for accumulating products
Output: $VALUE(V, P)$ $x \leftarrow 0$;

```
FOR $\gamma \in \Gamma(V)$ DO
BEGIN
$y \leftarrow 1$;
FOR $p \in P$ DO $y \leftarrow y \cdot [\gamma]$;
$x \leftarrow x + y$;
END;
OUTPUT ($x$)
```

ALGORITHM 2. *Generalized backtracking.*

Input parameter: n, a structure tree node for formula (V,P).
Global variables: v in V.
Local variables: x for accumulating a sum, y for accumulating products.
Assumptions: When called, global variables from $BV(n) - A(n)$ contain an assignment γ_0.
Value returned: $E(n, \gamma_0)$ from Definition 3.5. Returns $VALUE(V, P)$ when n is the root node.
Optimization: If ever $y = 0$, the program could skip to the next assignment α. A similar remark applies to other algorithms in this appendix.

```
    FUNCTION EVALUATE($n$)
$x$, $y$ are local variables.
$x \leftarrow 0$;
FOR $\gamma \in \Gamma(A(n))$ DO
BEGIN
$y \leftarrow 1$;
```

```
FOR $p \in B(n)$ DO $y \leftarrow y \cdot p[\gamma + \gamma_0]$;
FOR ALL CHILDREN $n'$ of $n$ DO $y \leftarrow y \cdot EVALUATE(n')$;
$x \leftarrow x + y$;
END;
RETURN(x);
```

ALGORITHM 3. *Backtracking with table lookups.*

Input parameter: n, a structure tree node for formula (V, P).

Global memory: variables v in V and, for each structure tree node m, a table T_m having an entry for each assignment γ in $\Gamma(CV(m))$. Table entries can be semiring elements or "undefined."

Local Variables: x for accumulating a sum, y for accumulating products.

Assumptions: When function is called, global variables from $CV(n) - A(n)$ contain an assignment γ_0. All table entries are initialized "undefined."

Value returned: $E(n, \gamma_0)$ from Definition 3.5. Returns $VALUE(V, P)$ when n is the root node.

```
FUNCTION $EVALUATE(n)$
$x$, $y$ are local variables.
IF $T_n(\gamma_0)$ = "undefined" THEN
BEGIN
$x \leftarrow 0$;
FOR $\gamma \in \Gamma(A(n))$ DO
BEGIN
$y \leftarrow 1$;
FOR $p \in B(n)$ DO $y \leftarrow y \cdot p[\gamma + \gamma_0]$;
FOR ALL CHILDREN $n'$ of $n$ DO $y \leftarrow y \cdot EVALUATE(n')$;
$x \leftarrow x + y$;
END;
$T_n(\gamma_0) \leftarrow x$;
END
RETURN $(T_n(\gamma_0))$
```

ALGORITHM 4. *Dynamic programming.*

Input: A structure tree for formula (V, P).

Memory: Variables x and y and, for each node m of the structure tree, a table T_m having one entry for each assignment in $\Gamma(CV(m) - A(m))$. The table entries are semiring elements.

Assumptions: Each table is initially considered "unconstructed" and is reclassified as "constructed" at the time indicated in the program.

Output: The table T_{root} for the root node has only one entry because $CV(root) - A(root) = \emptyset$ and thus on exit, this entry contains $VALUE(V, P)$.

```
WHILE not all $T_n$ are constructed DO
BEGIN
PICK $n$ such that
(1) $T_n$ is not constructed and
(2) $T'_n$ is constructed for all children of n;
FOR all $\gamma_0 \in \Gamma(CV(n) - A(n))$ DO
BEGIN
```

```
$x \leftarrow 0$;
For all $\gamma_1 \in \Gamma(A(n))$ DO
BEGIN
$y \leftarrow 1$;
FOR $p \in B(n)$ DO $y \leftarrow y \cdot p[\gamma_0 + \gamma_1]$;
For all children of $n'$ of $n$ DO $y \leftarrow y \cdot T_{n'}[\gamma_0 + \gamma_1$ projected to $CV(n') - A(n')]$;

                $x \leftarrow x + y$;
END;
$T_n[\gamma_0] \leftarrow x$
END;
{$T_n$ is now constructed};
END;
```

ALGORITHM 5. *Optimization using generalized backtracking.*

Input parameter: n, a structure tree node for formula (V,P).

Global variables: v in V.

Local variables: x for accumulating a maximum, y for accumulating sums, and v' for $v \in AD(n)$ for remembering the "best assignment for $AD(n)$ seen so far."

Assumptions: When called, global variables from $BV(n) - A(n)$ contain an assignment γ_0.

Value returned: $E(n, \gamma_0)$ from Definition 3.5. Additionally, the global variables $v \in AD(n)$ contain a corresponding assignment.

```
    FUNCTION $EVALUATE(n)$
$x$, $y$, $v'$ for $v \in AD(n)$ are local variables
$x \leftarrow \infty$;
For $\gamma \in \Gamma(A(n))$ DO
BEGIN
$y \leftarrow 0$;
FOR $p \in \beta(n)$ DO $y \leftarrow y + p[\gamma + \gamma_0]$
FOR ALL CHILDREN $n'$ of $n$ DO $y \leftarrow y + EVALUATE(n')$;
IF $y = x$ THEN {"IF $y < x$" is better}
BEGIN
$x \leftarrow y$;
FOR $v \in AD(n)$ DO $v' \leftarrow v$;
END;
END;
FOR $v \in AD(n)$ DO $v \leftarrow v'$;
RETURN $(x)$;
```

REFERENCES

[1] A. ARNBORG, D. G. CORNEIL, AND A. PROSKUROWSKI, *Complexity of finding embeddings in a k-tree*, SIAM J. Algebraic Discrete Meth., 8 (1987), pp. 277–284.

[2] S. ARNBORG AND A. PROSKUROWSKI, *Linear time algorithms for NP-hard problems restricted to partial k-trees*, Discrete Appl. Math., 23 (1989), pp. 11–24.

[3] S. ARNBORG, J. LAGERGREN, AND D. SEESE, *Which problems are easy for tree-decomposable graphs?*, J. Algorithms, 12 (1991), pp. 308–340.

[4] A. Bertoni, G. Mauri, and N. Sabadini, *A characterization of the class of functions computable in polynomial time on random access machines*, in Proc. 13th Annual ACM Symposium on the Theory of Computing, Association for Computing Machinery, New York, 1981, pp. 168–176.

[5] H. L. Bodlaender, *Dynamic programming on graphs with bounded tree width*, Technical Report RUU-CS-87-22, Department of Computer Science, University of Utrecht, Utrecht, the Netherlands, 1987; Lecture Notes in Comput. Sci., 317 (1988), pp. 105–118.

[6] ———, *NC-algorithms for graphs with small treewidth*, Lecture Notes in Comput. Sci., 344 (1988), pp. 1–10.

[7] ———, *A linear-time algorithm for finding tree-decompositions of small treewidth*, in Proc. 25th Annual ACM Symposium on the Theory of Computing, Association for Computing Machinery, New York, 1993, pp. 226–234; SIAM J. Comput., to appear.

[8] S. A. Cook, *The complexity of theorem-proving procedures*, in Proc. 3rd Annual ACM Symposium on the Theory of Computing, Association for Computing Machinery, New York, 1971, pp. 151–158.

[9] M. E. Dyer and A. M. Frieze, *Planar 3DM is NP-complete*, J. Algorithms, 7 (1986), pp. 174–184.

[10] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theorem of NP-Completeness*, W. H. Freeman, San Francisco, 1979.

[11] J. R. Gilbert, J. P. Hutchinson, and R. E. Tarjan, *A separator theorem for graphs of bounded genus*, J. Algorithms, 5 (1984), pp. 391–407.

[12] J. P. Hayes, *Digital simulation with multiple logic values*, IEEE Trans. Computer-Aided Design, 5 (1986), pp. 274–283.

[13] H. B. Hunt III, M. Marthe, V. Radhakrishnan, D. J. Rosenkrantz, and R. E. Stearns, *A unified approach for proving both easiness and hardness results for succinct specifications*, Technical Report 94-5, Computer Science Department, State University of New York at Albany, Albany, New York, 1994.

[14] H. B. Hunt III and R. E. Stearns, *The complexity of very simple Boolean formulas with applications*, SIAM J. Comput., 10 (1990), pp. 44–70.

[15] R. M. Karp, *Reducibility among combinatorial problems*, in Complexity and Computer Computations, R. E. Miller and J. W. Thatcher, eds., Plenum Press, New York, 1972, pp. 85–103.

[16] M. W. Krentel, *The complexity of optimization problems*, J. Comput. System Sci., 36 (1988), pp. 490–509.

[17] T. Lenguaer and K. W. Wagner, *The correlation between the complexities of nonhierarchical and hierarchical version of graph problems*, J. Comput. System Sci., 44 (1992), pp. 63–93.

[18] D. Lichtenstein, *Planar formulae and their uses*, SIAM J. Comput., 11 (1982), pp. 329–343.

[19] R. J. Lipton, D. J. Rose, and R. E. Tarjan, *Generalized nested dissection*, SIAM J. Numer. Anal., 16 (1979), pp. 346–358.

[20] R. L. Lipton and R. E. Tarjan, *Applications of a planar separator theorem*, SIAM J. Comput., 9 (1980), pp. 615–629.

[21] B. Monien and I. H. Sudborough, *Bounding the bandwidth of NP-complete problems*, Lecture Notes in Comput. Sci., 100 (1981), pp. 279–292.

[22] A. Panconesi and D. Ranjan, *Quantifiers and approximation*, in Proc. 22nd Annual ACM Symposium on the Theory of Computing, Association for Computing Machinery, New York, 1990, pp. 446–456.

[23] C. H. Papadimimitriou and M. Yannakakis, *The complexity of facets (and some facets of complexity)*, J. Comput. System Sci., 28 (1984), pp. 144–159.

[24] ———, *Optimization, approximation, and complexity classes*, J. Comput. System Sci., 43 (1991), pp. 425–440.

[25] V. Radhakrishnan, H. B. Hunt III, and R. E. Stearns, *Efficient algorithms for solving systems of linear equations and path problems*, in Proc. 9th Annual Symposium on Theoretical Aspects of Computer Science, Lecture Notes in Comput. Sci., 577 (1992), pp. 109–119.

[26] S. S. Ravi and H. B. Hunt III, *Application of planar separator theorem to counting problems*, Inform. Process. Lett., 25 (1987), pp. 317–321.

[27] B. A. Reed, *Finding approximate separators and computing treewidth quickly*, in Proc. 24th Annual ACM Symposium on the Theory of Computing, Association for Computing Machinery, New York, 1982, pp. 221–228.

[28] N. Robertson and P. D. Seymour, *Graph minors* II*: Algorithmic aspects of tree-width*, J. Algorithms, 7 (1986), pp. 309–322.

[29] ———, *Graph minors* XIII*: The disjoint path problem*, J. Combin. Theory Ser. B, 63 (1995), pp. 65–110.

[30] A. Rosenthal, *Dynamic programming is optimal for non-serial optimization problems*, SIAM J. Comput., 11 (1982), pp. 47–59.

[31] T. J. SCHAEFER, *The complexity of satisfiability problems*, in Proc. 10th Annual ACM Symposium on the Theory of Computing, Association for Computing Machinery, New York, 1978, pp. 216–226.

[32] R. E. STEARNS AND H. B. HUNT III, *Generalized satisfiability problems, structure trees, and their application*, Technical Report 90-2, Department of Computer Science, State University of New York at Albany, Albany, New York, 1990.

[33] ———, *Power indices and easier hard problems*, Math. Systems Theory, 23 (1990), pp. 209–225.

[34] L. VALIANT, *The complexity of enumeration and reliability problems*, SIAM J. Comput., 8 (1979), pp. 410–421.

[35] L. G. VALIANT AND V. V. VAZIRANI, *NP is as easy as detecting unique solutions*, in Proc. 17th Annual ACM Symposium on the Theory of Computing, Association for Computing Machinery, New York, 1985, pp. 458–463.

[36] A. T. WHITE, *Graphs, Groups and Surfaces*, McGraw–Hill, New York, 1984.

SIAM J. COMPUT.
Vol. 25, No. 3, pp. 477–497, June 1996

STRONGLY COMPETITIVE ALGORITHMS FOR PAGING WITH LOCALITY OF REFERENCE*

SANDY IRANI[†], ANNA R. KARLIN[‡], AND STEVEN PHILLIPS[§]

Abstract. What is the best paging algorithm if one has partial information about the possible sequences of page requests? We give a partial answer to this question by presenting the analysis of strongly competitive paging algorithms in the access graph model. This model restricts page requests so that they conform to a notion of locality of reference given by an arbitrary access graph.

We first consider optimal algorithms for undirected access graphs. Borodin et al. [*Proc. 23rd ACM Symposium on Theory of Computing*, 1991, pp. 249–259] define an algorithm, called FAR, and prove that it is within a logarithmic factor of the optimal online algorithm. We prove that FAR is in fact strongly competitive, i.e., within a constant factor of the optimum. For directed access graphs, we present an algorithm that is strongly competitive on *structured program graphs*—graphs that model a subset of the request sequences of structured programs.

Key words. analysis of algorithms, ònline algorithms, competitive analysis, paging, locality of reference

AMS subject classifications. 68Q22, 68R10

1. Introduction.

Many computer systems have a two-level store of memory consisting of a small, fast memory and a large, slow memory. The abstraction of a large, fast, virtual memory is often implemented by dividing the slow memory into pages and keeping those pages that are likely to be referenced soon in the fast memory. A *page fault* occurs when a reference is made to a page that is not resident in fast memory. Handling a page fault is typically expensive since the page must be brought into fast memory. A *page-replacement strategy* specifies which page to replace on a page fault.

If the future sequence of page requests is known in advance, the optimal page replacement strategy is clear: replace the page whose next request is farthest in the future [1]. Unfortunately, the decision about which page to replace must usually be made *online*, without detailed information about future requests.

How can we analyze such an online algorithm? Straightforward worst-case analysis is useless; if arbitrary request sequences are allowed, then an adversary that always requests the last discarded page can force any paging algorithm to fault on each request. Average-case analysis is also problematic since it requires a statistical model of the sequence of requests. It is extremely difficult to devise a realistic model, since the pattern of accesses changes dynamically with time and with different applications. Nonetheless, several of the early analyses of paging algorithms were performed assuming a fixed probability distribution on the request sequences [6, 13, 1].

In order to get around these problems, the notion of *competitive analysis* was introduced by Sleator and Tarjan [12]. An online page-replacement strategy A is said to be *c-competitive* if there exists a constant β such that for every request sequence

* Received by the editors August 19, 1992; accepted for publication (in revised form) September 15, 1994.

[†] Department of Information and Computer Science, University of California at Irvine, Irvine, CA 92717, (irani@ics.uci.edu). The research of this author was partially supported by National Science Foundation grant CCR-9309456.

[‡] DEC Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301.

[§] Department of Computer Science, Stanford University and DEC Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301. The research of this author was partially supported by National Science Foundation grant CCR-9010517 and an OTL grant.

σ,

$$A(\sigma) \leq c \cdot \mathrm{OPT}(\sigma) + \beta,$$

where $A(\sigma)$ is the cost (i.e., the number of faults) incurred by the algorithm A in processing the request sequence σ and $\mathrm{OPT}(\sigma)$ is the cost incurred by the optimal offline algorithm in processing σ. The *competitiveness* of A, denoted by c_A, is the infimum of c such that A is c-competitive.

Competitive analysis avoids the assumptions of probabilistic analysis, and has the power of differentiating between different paging algorithms. Sleator and Tarjan showed that no deterministic paging algorithm can achieve a competitiveness better than k, where k is the number of pages of fast memory. They also showed that some practical algorithms, such as first-in-first-out (FIFO) and least-recently-used (LRU), are k-competitive, and hence best possible in their model.

1.1. Locality of reference. The Sleator–Tarjan results conflict with practical experience on paging in at least two ways. First, FIFO and LRU have the same competitiveness, even though in practice LRU usually outperforms FIFO. Second, LRU usually incurs much less than k times the optimal number of faults, even though its competitiveness is k.

The reason for the practical success of LRU has long been known: most programs exhibit *locality of reference* [1, 4, 11]. This means that if a page is referenced, it is more likely to be referenced in the near future (temporal locality) and pages near it in memory are more likely to be referenced in the near future (spatial locality). Indeed, a two-level store is only useful if request sequences are *not* arbitrary.

Motivated by these observations, Borodin, Irani, Raghavan, and Schieber [2] proposed a technique for incorporating locality of reference into the traditional Sleator–Tarjan framework. Their notion of an *access graph* limits the set of request sequences the adversary is allowed to make.

An access graph $G = (V, E)$ for a program is a graph that has a vertex for each page that the program can reference. Locality of reference is imposed by the edge relation—the pages that can be referenced after a page p are just the neighbors of p in G or p itself. Thus a request sequence σ must be a walk on G. The specific walk that is generated is determined by the data given to the program. The definition of competitiveness remains the same as before, except for this restriction on the request sequences. Let $c_{A,k}(G)$ denote the competitiveness of an online algorithm A with k pages of fast memory on the access graph G. We denote by $c_k(G)$ the infimum (over online algorithms A with k pages of fast memory) of $c_{A,k}(G)$. Thus $c_k(G)$ is the best that any online algorithm can do. We say an algorithm A is *strongly competitive* if $c_{A,k}(G) = O(c_k(G))$.[1]

An access graph may be either directed or undirected. An undirected access graph might be a suitable model when the page reference patterns are governed by the data structures used by the program. For example, if a program performs operations on a tree data structure, and the mapping of the tree nodes to pages of virtual memory represents a contraction of a tree, then the appropriate access graph might be a tree. Alternatively, if we were to completely ignore data and focus only on the flow of control inherent in the structure of the program, a directed access graph might be a more suitable model.

[1] This terminology differs slightly from that of [7, 9], where strongly competitive was defined as achieving the competitive ratio $c_k(G)$.

Unfortunately, modeling access patterns is not as simple as these examples would suggest. See §4 for a discussion of the limitations of the model.

1.2. Summary of results. In this paper, we show that there are strongly competitive page replacement strategies in two important settings: (1) undirected access graphs and (2) directed access graphs representing the stream of instruction references made by a structured program.

1.2.1. Undirected access graphs. Borodin et al. proved that on any undirected access graph G, $c_{\mathrm{LRU},k}(G) \leq 2c_{\mathrm{FIFO},k}(G)$, and that LRU is often better than FIFO. However, on some graphs the competitiveness of both LRU and FIFO is much greater than $c_k(G)$. For example, it has been observed [8] that LRU and FIFO behave badly on loops just larger than k. This is substantiated by the following result in the access graph model: if G is the $(k+1)$-node cycle, then $c_k(G) = \lceil \log(k+1) \rceil$, while $c_{\mathrm{LRU},k} = k$. Thus LRU can be far from optimal among online algorithms.

We seek a *universal* algorithm, with a succinct description, whose competitiveness is close to $c_k(G)$ on *every* graph G. Importance is lent to this question by recent operating system research, in which facilities are provided for user-defined page replacement strategies [3, 10]. When these facilities are available, one must consider the following question: what is the best paging algorithm if one has partial information about the possible sequences of page requests? When the partial information is represented by an access graph, this question asks for a universal paging algorithm that is *strongly competitive.*

Borodin et al. describe a simple and natural algorithm called FAR, and prove that it achieves a competitiveness within $O(\log k)$ of $c_k(G)$for every graph G. They leave open the question of finding an algorithm with competitive ratio $O(c_k(G))$. Our main result for undirected access graphs is that FAR is optimal, within a constant factor, among online algorithms for paging with locality of reference.

THEOREM 1.1. *For any graph G and memory size k, the algorithm* FAR *is strongly competitive, i.e.,*

$$c_{\mathrm{FAR},k}(G) = O(c_k(G)).$$

The proof relies on a graph decomposition called a *vine decomposition* [2]. Borodin et al. show how to find a lower bound on $c_k(G)$ using a vine decomposition of G. We show that this lower bound is essentially optimal, by using the fault pattern of FAR to construct a vine decomposition of G.

1.2.2. Directed access graphs. In §3 we consider optimal algorithms for directed access graphs that represent a subset of the stream of instruction references made by a structured program. We use the notion of *structured program graphs* as in [2].

Structured program graphs (spg's) are defined recursively as follows: Every spg has a designated *start* node and a designated *stop* node. A single directed edge is an spg, where the node with outdegree 1 is the start node and the node with indegree 1 is the stop node. More complex spg's can be constructed by applying the following rules:

(i) Two spg's G_1 and G_2 can be combined to get a new spg, by identifying the start node of G_1 and the stop node of G_2 (serial composition).

(ii) Two spg's G_1 and G_2 can be combined to get a new spg, by identifying the stop nodes of G_1 and G_2 (branching statement).

(iii) A node v in an spg can be identified with a node u of a directed cycle to yield a new spg (loop).

Notice that there is no way to represent arbitrary "GOTO" statements. A significant limitation of this model is that the definition does not allow branching within loops.

Borodin et al. analyze a simple generalization of FAR, called 2FAR, that is optimal for spgs in which all strongly connected components have at most $k + 1$ nodes. We introduce a variant of 2FAR, called EVEN, and prove that it is strongly competitive for all spg's.

THEOREM 1.2. *The algorithm* EVEN *is strongly competitive on the class of structured program graphs. In other words, for any structured program graph G, $c_{k,\text{EVEN}}(G) = O(c_k(G))$.*

2. Undirected access graphs.

2.1. Background. We review some terminology and results on competitive paging. A sequence of page requests and the resulting behavior of a paging algorithm can be divided into *phases*. At the beginning of a phase, all pages are unmarked. When a page is *requested* (or *hit*) it is said to be *marked.* A phase ends just before the request to the $(k + 1)$st distinct node. At the end of a phase all nodes become unmarked again. We call the nodes requested in this phase but not in the previous phase *new* nodes. A node is said to be *evacuated* if it has been evicted from the fast memory during the phase. If a page is in the fast memory, we will say it is covered by a *server*.

The algorithm FAR is a *marking algorithm*. A marking algorithm is an algorithm that, on a fault, always evicts an *unmarked* page. The unmarked page that FAR chooses to evict is the page that is *furthest* (in the access graph) from the set of marked nodes.

We think of the sequence of page requests as being generated by an *adversary*, and say that the *cost* of the adversary is the cost of the offline algorithm on the request sequence.

PROPOSITION 2.1. (a) *If g_i new nodes are requested in the ith phase, then the cost of the adversary during the first i phases is at least $(\sum_{j=1}^{i} g_j)/2$, and at most $\sum_{j=1}^{i} g_j$.* (b) *If A is a marking algorithm then for any graph G, $c_{A,k}(G) \leq k$.*

The proof of part (a) is due to Fiat et al. [5], while part (b) is due to Karlin et al. [7].

A *vine decomposition* $\mathcal{V}(H) = (T, \mathcal{P})$ of any graph H is a tree T in H together with a set $\mathcal{P} = P_1, P_2, \ldots$ of pairwise edge disjoint simple paths in H (without loops), called *vines,* such that (i) the nodes of H are partitioned between T and the interior nodes of the P_i's; (ii) each endpoint of each path is a node in T. If a tree node is the endpoint of a vine, it is called an *anchor*. Let n_T be the number of leaves of T that are not anchors. The *value* of a path P, denoted $v(P)$, is defined to be $\lceil \log |P| \rceil$, where $|P|$ is the number of edges in P. The *value* of the vine decomposition $\mathcal{V}(H)$, denoted $v(\mathcal{V}(H))$ (or $v(\mathcal{V})$ for short), is

$$v(\mathcal{V}(H)) = \sum_i v(P_i) + n_T - 1.$$

Lastly, if C is a cycle on $k + g$ nodes, $1 \leq g \leq k$, then the *value* of C is $v(C) = \lfloor \log k - \log g \rfloor / 2$.

THEOREM 2.2. 1. *For any graph G and memory size k, let $\mathcal{V}$ be a vine decomposition on $k+1$ nodes in G. Then $c_k(G) \geq v(\mathcal{V})$.*

2. *Let G be a graph containing a cycle $\mathcal{C}$ on $k+g$ nodes, $1 \leq g \leq k$. Then $c_k(G) \geq v(\mathcal{C})$.*

Proof. The lower bounds are proved by describing an adversary who walks on the access graph, causing the online algorithm to fault a large number of times before the phase ends. This proves something a little stronger: that in *each phase*, the ratio of faults made by the online and offline algorithms is at least the stated lower bound.

The proof of part 1 is due to Borodin et al. [2]. For completeness we give an outline of the proof here. W.l.o.g. assume that G has $k+1$ nodes, so that the adversary incurs at most one fault per phase. An online algorithm A always leaves one node of G uncovered—this node is called the *hole*. The bound on $c_k(G)$ follows from noting that during a phase, the adversary can make A incur $v(\mathcal{V})$ faults. When A moves its hole to a new position following a fault, the adversary (and the request sequence) walks to that position, causing A to fault there. If the new location of the hole is a tree node, the adversary walks to the hole on a path that contains only marked nodes or internal tree nodes. If the new location of the hole is an interior node of a vine v, the adversary walks to the hole on a path that contains only marked nodes, internal tree nodes, anchors, and at most half the unmarked nodes on the vine v. The hole can be reached at least n_T times by requests to nonanchor leaves, and at least $v(P) = \lceil \log |P| \rceil$ times by requests to each vine P.

For part 2, assume w.l.o.g. that G is just a cycle $\mathcal{C}$ on $k+g$ nodes. There are g "holes" (nodes where the online algorithm A doesn't have a server) in G at any time. A phase starts with the adversary and A covering the same set of contiguous vertices. If A ever has a hole on a marked node, the adversary requests the path to the hole, passing only through marked nodes. The phase begins with the adversary issuing requests to all g of the uncovered nodes. At any point, the set of unmarked nodes forms a contiguous path. The adversary then repeats the following pattern: he requests all the nodes in whichever half of the path of unmarked nodes has more holes, causing the online algorithm to incur at least $g/2$ faults. When the pattern has been repeated $i = \lfloor \log k - \log g \rfloor$ times, the path of unmarked nodes has length $\lfloor k/2^i \rfloor < 2g$, and the adversary simply requests unmarked nodes till g remain, and the phase ends.

The adversary incurs only g faults during this process; at the start of the phase he vacates the g nodes that will remain unmarked at the end of the subsequence. Therefore the ratio between the number of faults incurred by the online algorithm and the adversary is $i/2 = \lfloor \log k - \log g \rfloor /2$. Since the situation at the end of this phase is the same as at the start, the adversary can repeat this behavior ad infinitum, hence $c_k(G) \geq \lfloor \log k - \log g \rfloor /2 = v(\mathcal{C})$. □

2.2. Proof of FAR's strong competitiveness. In this section we prove that FAR is a strongly competitive algorithm.

THEOREM 2.3. *For any undirected access graph G, $c_{\mathrm{FAR},k}(G) = O(c_k(G))$.*

For the entire proof, we restrict our attention to a single phase of the request sequence. Suppose that g new nodes are requested in the phase. By Proposition 2.1, the adversary incurs at least $g/2$ faults for this phase, amortized. The proof works by showing that there is a vine decomposition or large cycle whose value is at least c/g times the number of faults incurred by FAR in the phase, for some absolute constant c. The lower bound of Theorem 2.2 then implies that the number of faults incurred by FAR during the phase is $O(g \cdot c_k(G))$. Because this is true for all phases,

$c_{\mathrm{FAR},k}(G) = O(c_k(G))$.

We use the concept from [2] of *representatives* or *reps* for short. Each time FAR incurs a fault, it must vacate some node. Consider the sequence of nodes that are vacated during the phase: we divide this sequence into blocks of $g+1$ nodes that are vacated successively. The number of new nodes requested in the phase is g, so at any time during the phase, the number of nodes that have been evicted so far during the phase and are currently outside the fast memory is at most g. Therefore there is at least one node of each block that is marked before *any* node of the subsequent block is vacated by FAR. We pick such a node for each block and call it a *rep*.

Suppose that the number of reps for this phase is R. We need to construct a vine decomposition on $k+1$ nodes or a cycle whose value is $\Omega(R)$.

2.2.1. Preliminaries. Let G' be the directed subgraph of G containing every node that is marked during the phase, with a directed edge from u to v if the first request at v immediately follows a request at u. The graph G' is a directed tree, rooted at the first node requested during the phase, and a directed path from x to y in G' implies that x was marked before y.

The following definitions are fundamental to the rest of the proof.

DEFINITION 1. (i) *A node in G' is a* chain node *if it has degree* 2 *in G and both indegree and outdegree* 1 *in G'. Nodes that are not chain nodes are called* nonchain nodes. (ii) *A chain is a directed path in G' whose interior nodes are chain nodes, and whose endpoints are not chain nodes.*

There is a natural reason for not allowing a node whose degree is more than 2 in G to be a chain node in G'. The reason is that otherwise a chain might not be "self-contained" in the following sense: a chain node that had degree at least 3 in G could be close to a marked node outside the chain, but far from any marked node inside the chain. Excluding this case makes it possible to analyze chains in isolation.

We divide the reps into two types.

DEFINITION 2. *A rep is of type* 1 *if it is the last or one of the first five reps on a chain, or it is a nonchain node. Any other rep is of type* 2 *(i.e., any rep in the middle of a chain). Let R_1 and R_2 be the number of type-*1 *and type-*2 *reps, respectively.*

We have two constructions: The first is a vine decomposition which is a tree on $k+1$ nodes with $\Omega(R_1)$ leaves. The other construction is a vine decomposition $\mathcal{V}$ on $3k/2$ nodes whose vines have total value $\Omega(R_2)$. We use $\mathcal{V}$ to find a vine decomposition on $k+1$ nodes whose vines have total value $\Omega(R_2)$ or to find a large cycle whose value is $\Omega(R_2)$. Since one of R_1 and R_2 has value $\Omega(R)$, the two constructions will complete the proof of Theorem 2.3.

2.2.2. Constructing a vine decomposition with value $\Omega(R_1)$. We prove the following theorem.

THEOREM 2.4. *The graph G contains a subgraph on $k+1$ nodes which is a tree with $\Omega(R_1)$ leaves.*

Proof. Charge each type-1 rep that is on a chain to the far endpoint of the chain, and charge reps that aren't on chains to themselves. Each nonchain node is charged at most six times (because G' is a directed tree, so each node is the far endpoint of at most one chain). Hence the number of nonchain nodes is at least $R_1/6$. Now a nonchain node is either a leaf of G', the root of G', or has degree 3 or more in G. Let UG be the undirected version of the graph G'. Root the tree UG at the root of G'.

Case 1. At least half of the nonchain nodes in G' either

1. are leaves or the root in UG,
2. or have degree 3 or more in UG,

3. or have degree 2 in UG and a parent of degree 3 or more in UG.

If v satisfies condition 3, then its child in UG doesn't (and it has a child since it is not a leaf). And the first descendent of v that does not have degree 2 satisfies condition 1 or 2. Therefore if α nodes satisfy condition 3, then at least α nodes in the tree satisfy conditions 1 or 2. And so at least a quarter of the nonchain nodes in the tree satisfy conditions 1 or 2. Thus, at least $R_1/24 - 1$ nodes are leaves in UG or have degree 3 or more in UG.

The average degree of nodes in a tree is less than 2, since any tree with n nodes has $n-1$ edges. Therefore, at least half of the nodes not of degree 2 in UG are leaves, and UG is the desired tree.

Case 2. Otherwise, at least half of the nonchain nodes in G' have degree 3 or more in G, but degree 2 in UG and a parent of degree 2 in UG. Let S be a maximal set of vertices in UG that have degree 3 or more in G, degree 2 in UG, and whose parent in G' has degree 2 in UG and is not in S. It is easy to see that $|S| = \Omega(R_1)$. For each $v \in S$, let $a(v)$ be an arbitrary vertex that is a neighbor of u in G but not in UG.

For each $w \in G \setminus UG$, let $num(w) = |\{v | a(v) = w\}|$. For each such w, if $num(w)$ is 1 or 2, then add the node w to UG together with a single edge to one of the nodes $v \in UG$ such that $a(v) = w$. Thus w becomes a leaf of UG. Alternatively, if $num(w) \geq 3$, then add the node w to UG together with edges to all the nodes $v \in UG$ such that $a(v) = w$. For each of these nodes v, except for the one closest to the root of UG (breaking ties arbitrarily), sever the edge from v to its parent $p(v)$ in G'. Since $p(v)$ had degree 2 in UG, it becomes a leaf of UG, and UG remains a tree.

Since UG contains at most one new node for each node in S, at most $k/2$ nodes can be added to UG by this process. Finally, since at least $|S|/2$ leaves were added to UG, and $|S| = \Omega(R_1)$, UG has $\Omega(R_1)$ leaves.

We can then prune the tree until there are $k+1$ nodes, while keeping the number of leaves $\Omega(R_1)$. This can be done by first finding a node v whose removal from T results in components of size at most k. To find v, pick an arbitrary node u. Consider the tree rooted at u. If one of the children of u is the root of a subtree with more than k nodes, reset u to be that child. Keep resetting u until all the subtrees rooted at children of u have at most k nodes. Since the graph has at most $2k$ vertices, every time u is reset, the maximum number of nodes in a subtree rooted at a child of u decreases. Pick v to be the final u.

Gather the subtrees of v into groups as follows: consider each subtree and put it in the smallest numbered group possible so that no group has more than k nodes. There at most four groups, since the sum of the nodes in any two groups is more than k. One group must contain $\Omega(R_1)$ leaves of T. This group can be easily extended to a connected subgraph of T containing $k+1$ nodes, giving the desired tree. □

2.2.3. Constructing a vine decomposition with value $\Omega(R_2)$.

The first step towards finding a vine decomposition or cycle whose value is $\Omega(R_2)$ is to show that the separation between type-2 reps decreases exponentially along directed paths in G'.

DEFINITION 3. *For a chain node v, let $n(v)$ be the distance from v to the end of its chain.*

LEMMA 2.5. (i) *If v, w, and z are three consecutive reps on a chain, then $n(z) \leq n(v)/2$.*

(ii) *If y is the second-to-last rep on a chain in P and c is the third rep on a later chain in P, then $n(c) \leq n(y)$.*

(iii) *If P is a directed path in G', containing m type-2 reps, then $m \leq 2\lfloor \log l_1 \rfloor$, where l_1 is the length of the first chain in P containing a type-2 rep.*

Proof. Part (i) is proven as follows: Since v and w are consecutive reps on a chain, v is marked before w is evicted. Therefore, if w' is the last node on w's chain to be marked before w is evicted, then w' lies between v and w, and the unmarked portion of the chain extends from w' until the end of the chain at the time w is evicted. Note, however, that some of these nodes may not have servers on them. Since FAR evicts an unmarked node furthest from the set of marked nodes, if w is less than half the way from w' to the end of the chain, then every vertex between w and the midpoint must be a hole. Furthermore, none of them can be reps since any subsequent rep on the chain must be vacated after w is marked. Therefore, z must be beyond the midpoint of the chain from w' to the end, and so $n(z) \leq n(w')/2 \leq n(v)/2$.

For part (ii), suppose that z is the last rep on y's chain, and that a and b are the first two reps on c's chain (so the sequence of reps on chains in P is $\ldots, y, z, a, b, c, \ldots$). Since y is marked before z is evicted, z is at distance at most $n(y)$ from the set of marked nodes at the time it is evicted. Also, by part (i), $n(c) \leq n(a) - n(c)$. Now suppose that $n(c) > n(y)$. Then $n(a) - n(c) > n(y)$ also. But then the server at c is further from the set of marked nodes at the time z is evicted than z is, and would be vacated before z, so c cannot be a rep, a contradiction. So $n(c) \leq n(y)$.

For part (iii), let v_i, $1 \leq i \leq m$, be the sequence of type-2 reps on P. Let v_0 be the fifth type-1 rep on the first chain of P containing type-2 reps. By parts (i) and (ii) we have $n(v_{i+1}) \leq n(v_i)/2$ or $n(v_{i+2}) \leq n(v_i)/2$, for $0 \leq i \leq m-2$. Hence $n(v_m) \leq n(v_0)/2^{\lfloor \frac{m}{2} \rfloor}$. Since there are four type-1 reps on the first chain before v_0, we have by part (i) that $n(v_0) \leq l_1/4$. Therefore, $n(v_m) \leq l_1/(2^{\lfloor \frac{m}{2} \rfloor} + 2)$. Lastly, $n(v_m) \geq 1$, so $1 \leq l_1/(2^{\lfloor \frac{m}{2} \rfloor} + 2)$. Rearranging, we obtain $m \leq 2\lfloor \log l_1 \rfloor$. □

Because of this lemma, we know that the number of type-2 reps on any chain is logarithmic in the length of the chain. Therefore, if we could make all chains be vines in some vine decomposition, then the vine decomposition would have value $\Omega(R_2)$, and we'd be done. The problem is that in order to connect the vines, we may need to use some chains to construct the tree, leaving only a subset of the chains to become vines. The following procedure constructs a vine decomposition $\mathcal{V}$ (possibly containing a small number of additional nodes from outside G') for which the value of the chains that become vines is $\Omega(R_2)$.

The procedure keeps a contraction $\tilde{G}$ of G', where each edge of $\tilde{G}$ is part of a chain of G' that contains some type-2 reps.[2] It never contracts only part of a chain, so we can refer to a chain of G' that hasn't been contracted yet as a "chain of $\tilde{G}$" (though its endpoints in $\tilde{G}$ may be supernodes). The contraction $\tilde{G}$ is always a tree directed from the root, as contracting preserves this property.

The procedure repeatedly identifies a chain $\mathcal{C}$ of $\tilde{G}$ that will be a vine $\mathcal{V}$, and describes the path between the endpoints of $\mathcal{C}$ in the tree of $\mathcal{V}$. This path may contain some chains of $\tilde{G}$ (and hence G'), but we use Lemma 2.5 to show that $v(\mathcal{C})$ is at least a constant fraction of the number of type-2 reps on the path. The identified chain and the chains used in the path are then contracted, and the process continues.

[2] To *contract* an edge (u, v) in a graph means that u and v are merged into a *supernode*. Edges to and from u and v are replaced by edges to and from the supernode, with duplicates deleted. A contraction of a graph is obtained by repeatedly contracting edges.

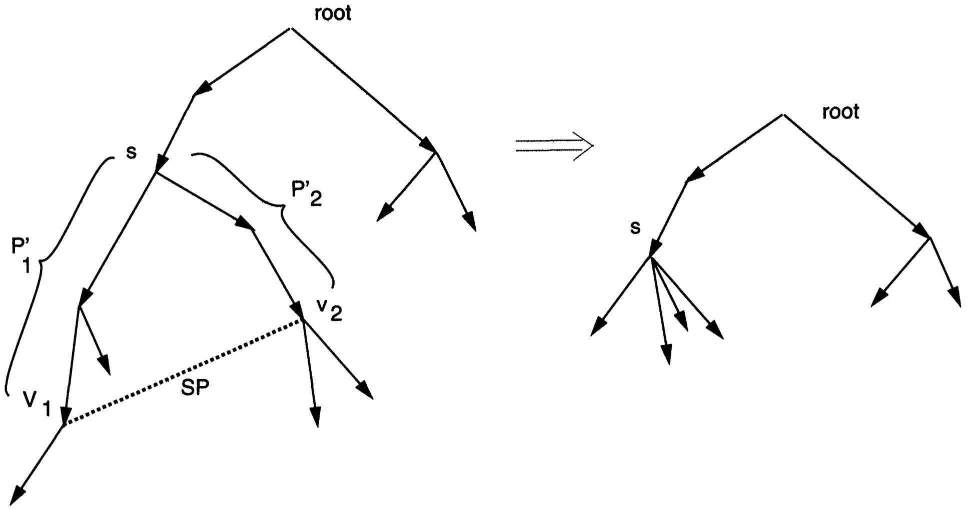


FIG. 1. *An iteration of the loop of* FindVineDecomp.

PROCEDURE FINDVINEDECOMP

Input: Undirected graph G, directed graph G' on nodes marked during the phase, and a record of FAR's faults during the phase.

Output: A vine decomposition $\mathcal{V}$ in G.

Initialization: First set $\tilde{G} = G'$. Then contract edges not in chains of G', and contract any chains that don't contain any type-2 reps. All edges so contracted are edges of the tree of $\mathcal{V}$. $\tilde{G}$ now consists of chains separated by (super)nodes.

Loop: Call the (super)node containing the first node marked in the phase *Root*. Repeat the following steps until *Root* is the only node in $\tilde{G}$.

1. Let v_1 and v_2 be two different (super)nodes in $\tilde{G}$. Let SP be the shortest path in G from any node contained in v_1 to any node contained in v_2 that doesn't use any edges of G' (if one exists). Choose v_1 and v_2 such that SP exists and $|SP|$ is minimal.

Let P_1 and P_2 be the directed paths in $\tilde{G}$ from *Root* to v_1 and v_2, respectively. Let s be the last supernode contained in both P_1 and P_2. Let P_1' be the portion of P_1 extending from s on, and similarly for P_2'. (See Figure 1. Edges in the picture correspond to chains in $\tilde{G}$.)

2. Now assume w.l.o.g. that P_1 has more type-2 reps than P_2. Let C be the first chain of P_1: C is identified as a vine of $\mathcal{V}$, and the edges in SP, P_1 (except for C), and P_2 are identified as edges in the tree of $\mathcal{V}$. In $\tilde{G}$ the paths P_1 and P_2 contracted (see Figure 1).

End FINDVINEDECOMP.

There must always be a pair of supernodes v_0 and v_1 for which SP exists, for the following reason. Let v be a leaf in $\tilde{G}$, and let C be the chain leading to v. If all paths from v to nodes marked earlier than those on C run through C, then there could be no more than one rep in C and v, by the definition of FAR—a contradiction.

The following lemma bounds the length of SP.

LEMMA 2.6. *In any iteration of the loop, the length of SP is at most $|C|/2$.*

Proof. The proof is similar to the proof of Lemma 2.5. Let x, y, and z be the last three reps (type 1 or 2) on the last chain of P_1. At the time y is evicted, x is marked, so at that time there must be some path Q of length at most $n(x) - n(y)$ from z, along the chain away from y, to earlier marked nodes (else z would be evicted before y). The path Q joins two different supernodes during this iteration of the loop and hence has a subpath that also joins two supernodes and has no edges of G'. Thus by minimality of $|SP|$ we have $|SP| \leq |Q| \leq n(x) - n(y)$.

Now from Lemma 2.5 we have $n(x) \leq |C|/2$, whence $|SP| \leq |C|/2$. □

LEMMA 2.7. *Procedure FindVineDecomp produces a vine decomposition $\mathcal{V}$ whose vines have value at least $R_2/4$, where R_2 is the number of reps of type 2. The number of nodes in $\mathcal{V} - G'$ is at most half the number of nodes in vines.*

Proof. Clearly the output of FindVineDecomp is a vine decomposition, $\mathcal{V}$. During each iteration through the loop, the number of (type-2) reps that become incorporated into a supernode is at most four times the value of the vine that is identified (this follows from Lemma 2.5, part (iii)). Hence the vines of $\mathcal{V}$ have value at least $R_2/4$.

The bound on the number of nodes of $\mathcal{V}$ follows from Lemma 2.6. □

Lastly we show how to use the vine decomposition $\mathcal{V}$ to construct either a large cycle or a vine decomposition on $k+1$ nodes, without losing much value.

THEOREM 2.8. *Either G contains a vine decomposition $\mathcal{V}'$ on $k+1$ nodes whose value is $\Omega(R_2)$, or a cycle on more than k nodes with value $\Omega(R_2)$.*

Proof. Let $\mathcal{V}$ be the vine decomposition given by procedure FindVineDecomp. We have two cases.

Case 1. $\mathcal{V}$ has a vine $\mathcal{C}$ with $r(\mathcal{C}) \geq R_2/16$. Suppose $\mathcal{C}$ has length l, and assume w.l.o.g. that $\mathcal{C}$ is contained in P_1. Let α be the length of the shortest cycle in G containing $\mathcal{C}$. If $\alpha \leq k+1$, the cycle can easily be extended to a vine decomposition with value at least $\log l$, while by Lemma 2.5, part (iii), $r(\mathcal{C}) = O(\log l)$, so $R_2 = O(\log l)$.

Otherwise $\alpha > k+1$. In addition, $\alpha \leq k(1+2^{-r(\mathcal{C})/4})$, since this is an upper bound on the length of the cycle containing SP, P_1, P_2, and some nodes inside supernodes (by Lemma 2.6). Hence by Theorem 2.2, part 2, the shortest cycle containing $\mathcal{C}$ has value at least $\lfloor r(\mathcal{C})/8 \rfloor = \Omega(R_2)$.

Case 2. No single vine $\mathcal{C}$ has $r(\mathcal{C}) \geq R_2/16$.

Let j be the number of nodes in vines. By Lemma 2.6, $\mathcal{V}$ has at most $k + j/2$ nodes, and by Lemma 2.7, $\mathcal{V}$ has value at least $R_2/4$. Delete the vines one by one, in decreasing order of length, till no more than $k+1$ nodes remain in $\mathcal{V}$. At most $j/2$ nodes are removed prior to the final deletion of vine $\tilde{v}$ that brings the number of remaining nodes below $k+1$. Since they are deleted in increasing order of ratio of value to number of nodes, the vine decomposition remaining before the deletion of $\tilde{v}$ has value at least $R_2/8$. The further deletion of $\tilde{v}$ leaves value at least $R_2/16$, since $\tilde{v}$ has value at most $R_2/16$. Lastly, we can easily extend the tree so that $\mathcal{V}$ has exactly $k+1$ nodes. □

2.2.4. Summary. We summarize the results of this section.

THEOREM 2.9. *For any graph G and memory size k, the algorithm* FAR *is strongly competitive, i.e.,*

$$c_{\mathrm{FAR},k}(G) = O(c_k(G)).$$

Proof. Let g be the number of new nodes requested in the current phase. By Proposition 2.1, the adversary incurs at least $g/2$ faults for this phase, amortized. Suppose that R is the number of reps for the phase; $2R$ is an upper bound on the competitive ratio of FAR for this phase. But $\Omega(R)$ is also a lower bound on $c_k(G)$. Indeed, either there are many type-1 reps, so we can find a tree with value $\Omega(R)$ using Theorem 2.4, or there are many type-2 reps, so we can use Theorem 2.8 to construct a vine decomposition on $k+1$ nodes or a cycle whose value is $\Omega(R)$. In either case, Theorem 2.2 gives $c_k(G) = \Omega(R)$. $\square$

3. Structured program graphs. The problem of devising competitive paging algorithms for directed access graphs is an important one. Unfortunately, in full generality this problem seems very difficult. There is, however, a restricted class of directed access graphs that is both important and tractable, which represents a subset of the stream of instruction references made by a structured program. We examine the class of *structured program graphs* (spg's) as defined in the introduction.

We assume that at run time, the paging algorithm knows the spg (the implications of this assumption are discussed in §4). The information that is not available until run time is the number of times a particular loop will be executed or the particular branch an execution path will take. In other words, it is unknown until run time exactly which path through the access graph will represent the access sequence for a particular execution. Fortunately, our analysis is worst-case over all paths through a particular access graph. The algorithm we introduce, called EVEN, is strongly competitive on all spg's. EVEN is a variation of an algorithm introduced in [2] called 2FAR.

Recall that spg's are recursively defined as follows: Every spg has a designated *start* node and a designated *stop* node. A single directed edge is an spg, where the node with outdegree 1 is the start node and the node with indegree 1 is the stop node. More complex spg's can be constructed by applying the following rules:

(i) Two spg's G_1 and G_2 can be combined to get a new spg by identifying the start node of G_1 and the stop node of G_2 (serial composition).

(ii) Two spg's G_1 and G_2 can be combined to get a new spg by identifying the start nodes and identifying the stop nodes of G_1 and G_2 (branching statement).

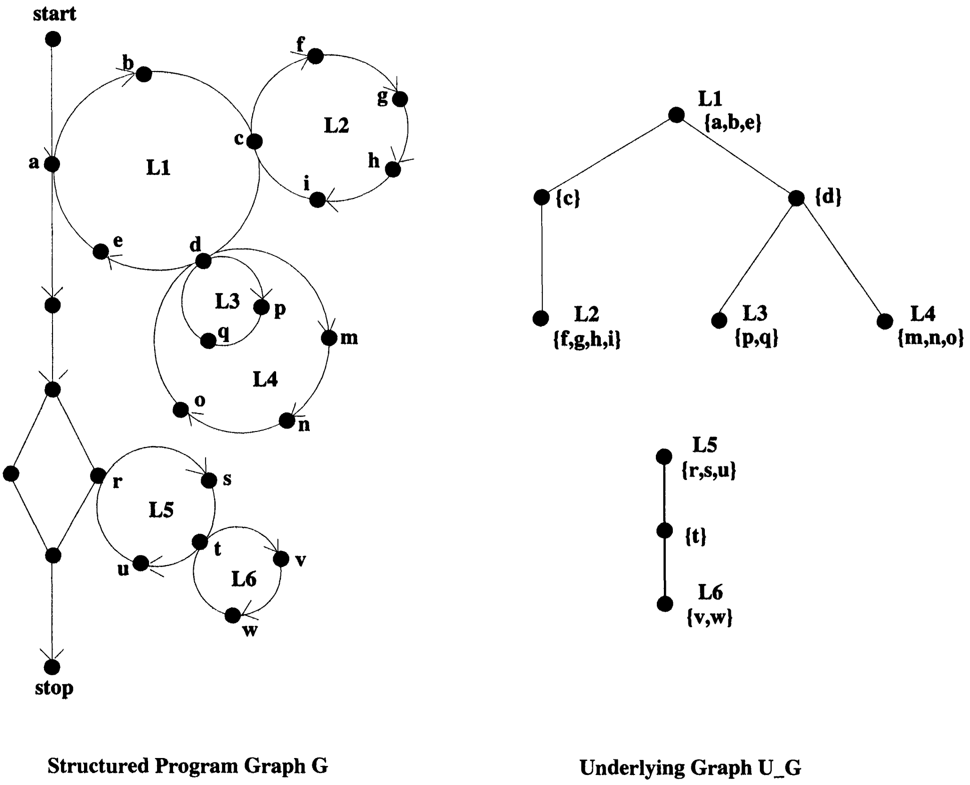
(iii) A node v in an spg can be identified with a node u of a directed cycle to yield a new spg (loop). We call v the *pivot* node for the cycle.

For each subgraph G' of an spg G (including the entire graph), we construct an undirected graph $U_{G'}$, called the underlying graph of G', that represents the relative embedding of the loops of G'. To distinguish between G' and U'_G, we call nodes in G' *nodes* and vertices in $U_{G'}$ *vertices.*

DEFINITION 4. *Let G' be a subgraph of an spg G. The* underlying graph $U_{G'}$ *of G' has one vertex for each loop l in G' (called a loop vertex), and one vertex for each pivot node in G' that is in at least two loops (called a pivot vertex). Each pivot vertex is adjacent to all the loop vertices corresponding to loops containing that pivot node. With each vertex u in $U_{G'}$, we associate a set of nodes of G', denoted $V(u)$: if u is a pivot vertex, then $V(u)$ is the corresponding pivot node. If u is a loop vertex corresponding to loop l, then $V(u)$ is the set of nonpivot nodes in l, and pivot nodes in l that are in only one loop.*

Figure 2 shows an example of an spg G and its underlying graph U_G. For each vertex v in the underlying graph, the nodes in $V(v)$ are shown in curly braces. Vertices in U_G with the name of a loop next to them are loop vertices; the remaining vertices are pivot vertices.

A strongly connected subgraph F of G is a subgraph such that for any two nodes w and v in F, there is a path from w to v and a path from v to w that only uses nodes and edges from F. Notice that if F is a strongly connected subgraph of an spg



Structured Program Graph G

Underlying Graph U_G

FIG. 2. *An spg and its underlying graph.*

G with more than one node and v is a vertex in F, then v is in a loop. In other words, $v \in V(u)$ for some $u \in U_G$.

LEMMA 3.1. *The underlying graph U_F of a strongly connected subgraph F of G is a tree. The underlying graph U_G of an spg G is a forest.*

Proof. By the construction rules for spg's, a strongly connected subgraph F of G consists of a connected set of cycles. Suppose that U_F contains a cycle. Since two loop nodes or two pivot nodes are never adjacent, every loop node contained in the cycle must be adjacent to two pivot nodes in the cycle. Among the loops represented in the cycle, let l be the last one added to the spg according to the construction rules above. Suppose that v and w are the two pivot nodes adjacent to l in the cycle. Since l was added last, a loop containing v and a loop containing w were already in the spg at the time l was added. But by the construction rule for loops, l can only be adjacent to one of them, which contradicts the existence of this cycle. Therefore U_F must be a tree.

Since U_G is the union of underlying graphs of maximal strongly connected subgraphs of G, and each of these is a tree, U_G must be a forest. □

3.1. A lower bound. A vertex in U_F is said to be a *peripheral vertex* if its removal does not disconnect U_F. If u is a peripheral vertex in U_F, $V(u)$ is said to be a *peripheral set* in F. The *size* of a peripheral set is its cardinality, $|V(u)|$. Note that if a node is in a peripheral set of F, then it is contained in only one loop in F. For any strongly connected graph F on $k + g$ nodes, define $l(F, g)$ to be the number of peripheral sets in F that have size at least g. Whenever the value of g is clear from context, we say that a peripheral set is *large* if it has size at least g, and we abbreviate $l(F, g)$ to $l(F)$.

We need the following preliminary lemmas.

LEMMA 3.2. *Let F be any $(k+g)$-node strongly connected subgraph of an spg. Let S be the set of peripheral sets in F. Then the following hold:*

1. *For any $s_1, s_2 \in S$, and for any nodes $x \in s_1$ and $y \in s_2$, there is a path in F from x to y that doesn't pass through any other peripheral set $s_3 \in S$, $s_3 \neq s_1, s_3 \neq s_2$.*
2. *For any y that is not in any peripheral set, and for any x, there is a path from x to y that doesn't contain nodes from any peripheral set (except perhaps for some nodes in s_1 such that $x \in s_1 \in S$).*

Proof. For the first part, consider the directed graph F' obtained from F by deleting, for each large peripheral set s_3 ($s_3 \neq s_1, s_3 \neq s_2$), all nodes in s_3. Since F' is still strongly connected, the desired path from x to y remains.

For the second part, consider the directed graph F' obtained from F by deleting, for each large peripheral set s ($s \neq s_1$ such that $x \in s_1$), all nodes in s. Since F' is still strongly connected, the desired path from x to y remains. □

The following theorem gives a lower bound on the competitive ratio achievable on an spg G. Let $\mathcal{H}_i(G)$ be the set of strongly connected subgraphs of G on i nodes. Let

$$comp(G) = \max_F \max\{(l(F)-1)/g, (H_{l(F)}-1)\},$$

where the maximum is taken over $F \in \mathcal{H}_{k+g}(G), k \geq g \geq 1$, and H_j is the jth harmonic number.

THEOREM 3.3. *If G is a structured program graph, then*

$$c_k(G) \geq comp(G).$$

Proof. Let F be any strongly connected subgraph of G on $k+g$ nodes, and let A be a paging algorithm. Let S be the set of large peripheral sets of F.

We now describe the adversary strategy. The adversary operates in phases (slightly different from the phases referred to elsewhere). We show that the adversary can force A to incur $g \cdot comp(G)$ faults in a phase while the adversary only incurs g faults in a phase. A phase starts with all of the adversary's holes on one set in S and the last request on a node in $\cup S$.

If the algorithm A ever maintains a hole on a node not contained in $\cup S$, or in a set in S that has already been requested in the phase, then the adversary requests a path to the hole without passing through any unrequested nodes in $\cup S$. Thus we can assume that all of A's holes are on unrequested nodes in $\cup S$. In this case, the adversary requests a path to the peripheral set in S with the most holes, such that the path followed doesn't hit any other peripheral sets in S. The phase ends when all but one of the large peripheral sets have been hit.

When the ith peripheral set is hit, A incurs at least $\lceil g/(|S|-i+1)\rceil$ faults. Therefore, the algorithm incurs at least

$$\begin{aligned}\lceil g/|S|\rceil + \lceil g/(|S|-1)\rceil + \cdots + \lceil g/2\rceil &\geq \max(g(H_{|S|}-1), |S|-1)\\ &\geq g \cdot comp(G)\end{aligned}$$

faults. Meanwhile, the adversary services the sequence by initially moving all g of its holes to the peripheral set that will not have been hit at the end of the phase and thus incurs at most g faults during the phase. □

3.2. The algorithm EVEN. We now explain the operation of EVEN on spg's. EVEN works in two modes. The first mode is when the set of nodes occupied by the servers and the present request are not contained in a strongly connected component. In this case, EVEN services a fault by using any server which is on a node that is not reachable from the current request. In the second mode, all the servers are in a strongly connected component and EVEN works as a marking algorithm. A new phase begins either when EVEN switches from mode 1 to mode 2 or when $k+1$ nodes have been marked in the previous phase (during the second mode). Roughly, EVEN attempts to vacate servers in peripheral node sets, and does so in a uniform way. A few definitions are necessary in order to rigorously describe the algorithm.

Let P be the set of nodes occupied by servers at the beginning of a phase. Let $\bar{P}$ be the smallest strongly connected subgraph that contains all of P.

Consider the underlying graph of $\bar{P}$, $U_{\bar{P}}$. Root $U_{\bar{P}}$ at the vertex whose node set contains the last request of the previous phase. This vertex is unique. As noted earlier, each vertex u in the underlying graph of $\bar{P}$, $U_{\bar{P}}$, corresponds to a node set in $\bar{P}$, $V(u)$. A node set is *empty* if it does not have any servers on it. A *limb* $L = (l_1, \ldots, l_r)$ in $U_{\bar{P}}$ is a simple path in $U_{\bar{P}}$ such that l_1 is a peripheral vertex (leaf of $U_{\bar{P}}$) and l_{i+1} is the parent of l_i for $1 \leq i < r$. The *limb set* $V(L)$ is the set of nodes corresponding to the vertices of the limb L (i.e., $\cup_{1 \leq i \leq r} V(l_i)$). EVEN keeps track of the limbs, making sure that each limb has servers only on the nodes in $V(l_r)$, i.e., all but the last vertex on the limb is empty. The vertex l_r (the endpoint of the limb that is furthest from the leaf) is called the *active vertex*, and $V(l_r)$ is called the *active set.*

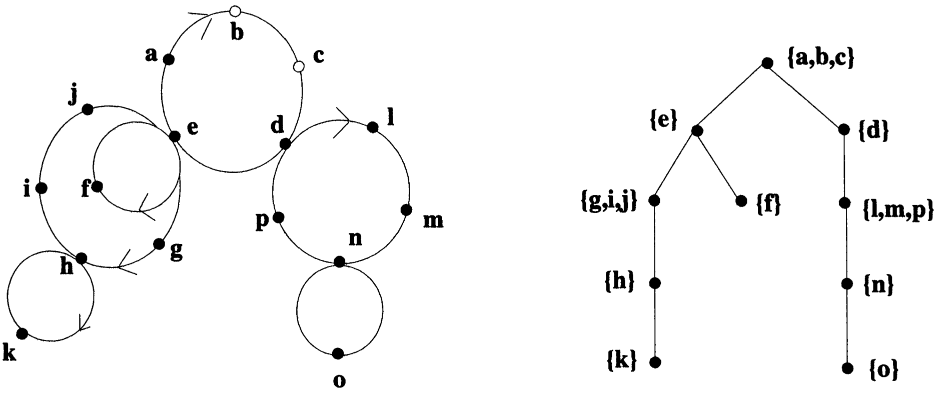
Limbs grow toward the root during the phase. At the beginning of a phase, each peripheral vertex is a limb. The peripheral set is the active set for that limb. Notice that because we chose $\bar{P}$ to be the minimum strongly connected subgraph that contains all nodes with servers, every peripheral set will contain at least one server. EVEN will always evacuate servers from the active set of some limb. A limb is said to *die* when any of the following conditions hold:

1. The active node set empties and some node in the node set of the parent vertex has been requested.
2. The active node set empties and the parent vertex has another child that is not yet in a dead limb.
3. A node in the active set is requested.

A *live* limb is one that has not died. The *value* of a limb L is the number of nodes in the limb set $V(L)$ that have been evacuated during the phase.

EVEN's policy for deciding which node to vacate when servicing a fault is determined as follows: Pick the live limb of minimum value. Evacuate any node in the node set of the active vertex. If the active vertex becomes empty then add the parent vertex to the limb (unless the limb dies for the reasons described above). The parent becomes the limb's active set. If there are no live limbs, then evacuate the unmarked node that is occupied by a server and is farthest from the current request.

Figure 3 shows an example of the operation of EVEN in mode 2. On the left side of the figure is the subgraph $\bar{P}$, and on the right side is the corresponding underlying graph $U_{\bar{P}}$. At the beginning of the phase all nodes are occupied by servers except for b and c, and the last request of the previous phase was at a. Initially there are three limbs, the peripheral vertices of the underlying graph: $\{k\}$, $\{f\}$, and $\{o\}$. The request to b results in the eviction of o, at which point, the parent vertex is added to the limb, resulting in the limb $(\{o\}, \{n\})$, with active vertex $\{n\}$. The request to c results in the eviction of f, at which point we observe that the parent of $\{f\}$ $(\{e\})$



Last Request of Previous Phase: a

Request Sequence: b,c,d,e,g,h,i,j,e,f,e,g,h,k,h,i,j,e,a,b,c,d,l,m,n,o

FIG. 3. *An example $\bar{P}$, $U_{\bar{P}}$, and request sequence.*

in the underlying graph has another child ($\{g, i, j\}$) that is not yet in a dead limb, and so by rule 2, the limb $\{f\}$ dies. At this point the two holes are at f and at o. Therefore, no further faults are incurred until the request to f, at which point k is evicted. The parent vertex of $\{k\}$ (h) has already been requested, and so by rule 1, the limb $\{k\}$ dies. The next fault is at the request to k, which results in the eviction of n, and the only remaining limb is extended to ($\{o\}, \{n\}, \{l, m, p\}$), with active set $\{l, m, p\}$. At this point, the two holes are at o and n. Finally, when l is requested, the limb dies by rule 3, since a node in the active set is requested.

In the next section, we bound the competitive ratio of EVEN.

3.3. The upper bound.

We will need the following three lemmas.

LEMMA 3.4. *Consider a phase of* EVEN. *If there are no live limbs, but there remain unmarked nodes that are occupied by servers, then all of the unmarked servers reside on the only simple path from the currently requested node to the first node that was requested in the phase.*

Proof. If no live limbs remain, then every loop with unmarked servers in $\bar{P}$ has a node that has been requested in the phase. Consider the set S of all nodes in $\bar{P}$ which have not been requested during the phase but are contained in cycles where some node has been requested. All unmarked servers reside on nodes in this set. Furthermore, since the set of requested nodes is contiguous, S forms a simple connected path in $U_{\bar{P}}$ from the most recently requested node to the first node requested in the phase. □

LEMMA 3.5. *At any point in a phase of* EVEN, *if there are l live limbs, the value of each limb is bounded above by $\lceil g/l \rceil$, where g is the number of new nodes requested in the phase.*

Proof. The values of any two live limbs differ by at most 1. Furthermore, the total number of evacuated nodes in live limbs at any time is at most g. The lemma follows. □

LEMMA 3.6. *Consider a phase of* EVEN *where $|\bar{P}| = k + g$. Let L be the set of limbs that grows in the phase. Then the limbs in L can be divided into two sets, L_1*

and L_2, and there is a strongly connected subgraph H of $\bar{P}$ of size $k+\alpha$, $1 \leq \alpha \leq g$, such that the following hold:

1. *Every peripheral set in H has size at least α.*
2. $|L_1| \leq 2l(H,\alpha)$.
3. *The total number of nodes contained in limbs in L_2 and evacuated during the phase is at most $2(g-\alpha)$.*

Proof. We trim $\bar{P}$ by repeatedly applying the following procedure: Let R be the subgraph of $\bar{P}$ that remains (initially $R = \bar{P}$). While R has $k+h$ nodes, but has peripheral sets of size less than h, apply the following rules: Pick a peripheral vertex r of U_R such that $V(r) < h$. Remove r from U_R and remove $V(r)$ from R. (Note that since every leaf of U_R is a loop vertex, R remains strongly connected.) If as a result of this operation there is a peripheral vertex v that is a pivot vertex, remove v from U_R and set $V(p(v))$ to be $V(p(v)) \cup V(v)$, where $p(v)$ is the parent of v in U_R. (Note that $p(v)$ is again a loop vertex.)

Call the resulting graph H. By the termination condition of the procedure, if H has $k+\alpha$ nodes, then every peripheral set has size at least α.

Figure 4 shows an example of the trimming operation. On top we see $\bar{P}$ and $U_{\bar{P}}$, with thick lines on the underlying graph representing the limbs. During the trimming process, loops a, b, c, and d are trimmed. The resulting underlying graph, U_H, is shown in the bottom left. Note that in the trimming process, limb l_4 was removed completely, whereas only part of limb l_2 was removed. Note that the intersection of l_2 with H does not contain a peripheral set of H.

Returning to the remaining two conditions of the lemma, if a limb in L has no intersection with H, then put it in L_2 and remove it from L. The total number of nodes in the limbs placed in L_2 so far is at most the total number of nodes removed from $\bar{P}$, which is in turn upper bounded by $g-\alpha$.

All remaining limbs in L intersect H. To account for these, create a limb tree as follows: Create a special root node. In addition, for each limb, create a node in the limb tree. Limb l is the parent of limb l' if the highest vertex in l' is the child of a vertex in l in the underlying graph $U_{\bar{P}}$. Any remaining limbs without parents take the root to be their parent. Note that any leaf in the limb tree corresponds to a limb l that contains a peripheral set in H. Other limbs may or may not contain peripheral sets of H. Figure 4 shows an example limb tree.

Let L' be the set of nodes in the limb tree whose parent has degree 2 (and is not the root). Then $|L'|$ is the number of degree-2 limb nodes in the tree. Therefore, there are at least $|L| - |L'|$ nondegree-2 limb nodes. But the average degree of nodes in a tree is less than two, and so at least $(|L| - |L'|)/2$ limbs in the tree are leaves and hence contain a peripheral set in H. We place the limbs in $L \setminus L'$ in L_1. Therefore, the total number of limbs in L_1 is at most $2l(H,\alpha)$ so far.

Consider a limb l in L'. Let $p(l)$ be the limb corresponding to the parent of l in the limb tree.

If $p(l)$ contains a peripheral set of H, add l to L_1. This limb can be amortized against the peripheral set belonging to $p(l)$. Note that $p(l)$ is not a leaf of the limb tree, and therefore is not charged against by L_1 limbs from $L \setminus L'$. ($p(l)$ may or may not end up in L_1 itself, depending on whether it is in L' or not.) Consequently, the total number of limbs in L_1 remains at most $2l(H,\alpha)$.

If $p(l)$ does not contain a peripheral set in H, then place l in L_2. We just have to prove that the number of evacuated nodes in $p(l) \cap \{\bar{P} \setminus H\}$, which we denote by S, upper bounds the number of evacuated nodes in l. (A node is said to be *evacuated*

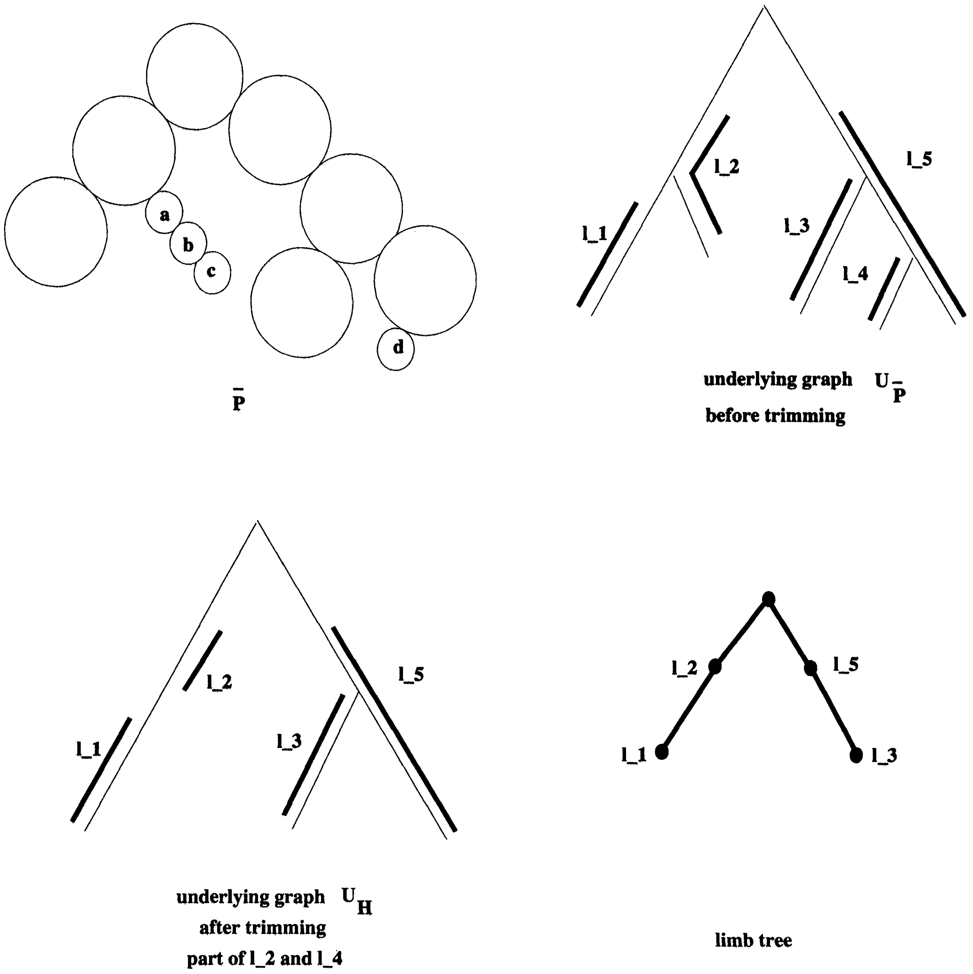


FIG. 4. *An example of the trimming process and limb tree.*

if it was evacuated some time in the phase). Since l is the only child of $p(l)$, we are amortizing against the set S only once.

Let t be the highest vertex in l. Let T be the parent of t and t' the only sibling of t in $U_{\bar{P}}$. Since the limb $p(l)$ has only one child in the limb tree and does not contain a peripheral set in H, t' and all of its descendants in $U_{\bar{P}}$ do not appear in U_H. That is, the set of nodes in $p(l)$ corresponding to t' and its descendants is exactly S. Recalling that the loop T is part of the limb $p(l)$ (otherwise $p(l)$ wouldn't be the parent of l in the limb tree), it must be that l died before $p(l)$ died. This means that t became empty before t' became empty and so the number of evacuated nodes in l is at most $|S|$. □

THEOREM 3.7. *The algorithm* EVEN *is strongly competitive on the class of structured program graphs. In other words, for any structured program graph G, $c_{k,\mathrm{EVEN}}(G) = O(comp(G))$.*

Proof. Consider an spg G. If EVEN incurs a fault on node v when in mode 1, then it is the first time that node v has been requested during the entire sequence. As a result, both EVEN and the optimal algorithm fault on the request to node v.

Thus, when EVEN is in mode 1, the number of faults that it incurs is equal to the number of faults the optimal algorithm incurs. We must now bound the number of faults incurred by EVEN in mode 2.

Let P' be the set of nodes requested in the phase before the previous phase. Let P be the set of nodes requested in the previous phase and let C be the set of nodes requested in the current phase. If the algorithm just switched from mode 1 to mode 2, then P is the set of nodes currently occupied by the servers, and P' are the k distinct nodes requested previous to the first request to any node in P.

Let g' be the number of new nodes requested in the last phase ($g' = |P - P'|$) and let g be the number of new nodes requested in this phase ($g = |C - P|$). If the number of faults EVEN incurs for an arbitrary phase is bounded by $O((g+g')c_k(G))$, then by Proposition 2.1, $c_{k,\text{EVEN}}(G) = O(c_k(G))$.

If $P' \cap C = \emptyset$, then $g' + g \geq k$. In this case, EVEN incurs $O(g' + g)$ faults during the phase, since no marking algorithm incurs more than k faults in a phase. If $P' \cap C \neq \emptyset$, then there is a strongly connected component containing P, contained in $C \cup P \cup P'$. Indeed, there is a directed path from the first node requested in C to a node x in $P' \cap C$. (This path contains at most g nodes outside P.) Since $x \in P'$, there is a path from x to the first node requested in the previous phase. (This path contains at most g' nodes outside P.) Finally, there is a path entirely inside P from the first node requested in the previous phase to the first node requested in the current phase. Therefore, if $\bar{P}$ is the smallest strongly connected subgraph containing P, $\bar{P}$ has at most $k + g + g'$ nodes.

Let L be the set of limbs in $\bar{P}$ that evolve in the phase. We now apply Lemma 3.6 to obtain a graph H of size $k + \alpha$ and a partition of L such that:

1. Every peripheral set in H has size at least α.
2. $|L_1| \leq 2l(H, \alpha)$.
3. The total number of nodes contained in limbs in L_2 that are evacuated during the phase is at most $2(g + g')$.

We can now show that the number of faults that EVEN incurs in the current phase is $O(g + g' + l(H) + gH_{l(H)})$. We count the number of faults by counting the number of nodes evacuated in $\bar{P}$. In fact, we only count the number of nodes in L that are evacuated. This is sufficient because Lemma 3.4 implies that there are at most $g + g'$ faults inbetween the time that the last limb dies and the end of the phase. As long as there are live limbs, all evacuated nodes are contained in limbs. Thus, the number of nodes that are not in any limb in L and are evacuated during the phase is at most $g + g'$.

We now bound the number of nodes in L that are evacuated. The number of nodes in L_2 evacuated in the phase is at most $2(g+g')$, by condition 3 in Lemma 3.6.

The number of evacuated nodes that are on limbs in L_1 is just the sum over all limbs in L_1 of the value of that limb when it dies. Consider the ith limb in L_1 that dies. By Lemma 3.5, the number of evacuated nodes on that limb is at most $\lceil g/(|L_1| - i + 1)\rceil$. Thus by condition 2 of Lemma 3.6, the total number of evacuated nodes in L_1 is at most

$$\sum_{j=1}^{2l(H)} \lceil (g/j) \rceil = O(l(H) + gH_{l(H)}).$$

We have shown that the number of faults that EVEN incurs in the current phase is $O(g + g' + l(H) + gH_{l(H)})$. Since $\alpha \leq g + g'$ and, by Theorem 3.3, $l(H)/\alpha + H_{l(H)} \leq$

$c_k(G)$, the number of faults EVEN incurs in the phase is

$$O(g+g'+l(H)+gH_{l(H)}) = O\left((g+g')\left(\frac{l(H)}{\alpha}+H_{l(H)}\right)\right) = O((g+g')c_k(G)). \qquad \square$$

4. Limitations of the model and open problems. There are four fundamental limitations to our work. The first is the fact that the model of structured program graphs that we use does not allow branching within loops.

The second limitation concerns our assumption that the paging algorithm knows the access graph. This is not realistic for several reasons. If the set of virtual pages accessed by a structured program and the data on those pages were static, life would be easy. In fact, the virtual pages associated with the data and instructions of a program typically fall into three categories.

(i) *Static.* At link time, program text (code) and global variables are assigned virtual addresses. These addresses remain fixed for the lifetime of the program.

(ii) *Heap.* Dynamically allocated data are assigned virtual addresses at runtime, when the memory is allocated. As long as these data are not deallocated, their virtual addresses remain fixed. However, once this storage is reclaimed, these virtual addresses can be allocated to other data. Consequently, the data on virtual pages associated to the heap may change many times over the lifetime of the program.

(iii) *Stack.* Variables local to procedures are assigned virtual addresses on the stack. Since the stack is growing and shrinking as the program runs, in accordance with the set of procedures being executed, a specific procedure's local variables may have several different virtual addresses over the runtime of the program.

Consequently, a given virtual address may store many different data items, and a given data item may reside in several different virtual addresses. This implies at the very least that one cannot directly associate program data with specific virtual pages.

Nonetheless, access graphs have well-defined semantics: an edge from virtual page p_1 to virtual page p_2 in an access graph means that at some point in the virtual address trace of the program, a reference to an address on p_2 immediately follows a reference to an address on p_1.

Since the access graph depends on the virtual address trace of the program, which in turn depends on the input data, it is a nontrivial problem to construct it. (In fact, in its most general form, the problem is undecidable since one must know if the program will halt in order to determine the access graph.) In practice, if one wishes the access graph to be accurate, one may need to execute or simulate the execution of the program.

This limitation raises several interesting open questions: Are there good approximations to the access graph that can be determined at compile time? Is there a strongly competitive algorithm that "learns" the access graph as more requests are seen? Also, are there good algorithms that only maintain a portion of the access graph, say, the subgraph generated by the pages in fast memory?

A third fundamental limitation of this model is that there is only a single pointer into the access graph at any time. For example, the spg's that we have studied only make sense in the context of the instructions of a program. Spg's do not adequately model both the instruction and the data references of a program. In order to model both, one would need at least two pointers into the access graph, one into a structured program portion of the graph, and another into an undirected or directed subgraph whose pages are storing program data. The request sequence would then consist of a walk on the graph, where the next request is always a neighbor of one of the two nodes pointed to. One could also imagine having more than two pointers, for example, if there were a collection of pointers into arrays, or other data structures in the program. It remains an interesting open question to generalize our results to the multiple-pointer case.

The last fundamental limitation of our work is that the competitive analysis we employ is worst-case. We optimize the worst-case ratio between our algorithm and the optimal offline, assuming that an adversary chooses the request sequence. Locality of reference is enforced by limiting the adversary's sequences to walks on the access graph. Nonetheless, walks on the access graph may bear little resemblance to the program address trace from which the access graph was generated. Therefore it is not obvious that optimizing the competitive ratio against such an adversary yields an algorithm that performs well in practice.

Recently, Karlin, Phillips, and Raghavan attempted to address this problem by studying paging algorithms for the case where the request sequence forms a Markov chain. The Markov chain incorporates locality of reference into the sequence, while eliminating the whole notion of an adversary.

The theoretical work on paging with access graphs could be complemented very well by experimental studies. It would be interesting to evaluate whether there are approximations to access graphs that model realistic page request sequences well. It would also be extremely interesting to evaluate how well the competitive algorithms described in this paper perform in practice.

Another interesting direction for research relates to granularity. In the introduction, we mentioned that an undirected access graph might be a suitable model for a program that performs operations on a tree data structure, as long as the mapping of the tree nodes to pages of virtual memory represents a contraction of the tree. Similarly, spg's are good models of the flow of control among individual program statements. However, once the instructions are assigned virtual addresses, the underlying access graph may not look anything like an spg, since many instructions are assigned to a single virtual page.

For a fixed page replacement strategy, different assignments of virtual addresses to data and procedure entry points can result in vastly different page-fault rates, since changing the layout changes the partition into pages. It would be very interesting to study techniques for laying out instructions and data in virtual memory, so that the partition into virtual pages results in a low page-fault rate.

Finally, dealing with the most general definition of structured program graphs remains open, as well as two questions from [2]:

Open Question 1. Design an algorithm that is strongly competitive on all structured program graphs (i.e., allow branching within loops).

Open Question 2. Show that for all G and k, $c_{\mathrm{LRU},k}(G) \leq c_{\mathrm{FIFO},k}(G)$.

Open Question 3. Is there a "universal" randomized algorithm that is close to optimal on every G? Is there a graph-theoretic lower bound on $c_k(G)$ for randomized algorithms against an oblivious adversary?

REFERENCES

[1] L. A. BELADY, *A study of replacement algorithms for virtual storage computers*, IBM Systems J., 5 (1966), pp. 78–101.

[2] A. BORODIN, S. IRANI, P. RAGHAVAN, AND B. SCHIEBER, *Competitive paging with locality of reference*, J. Comput. System Sci., 50 (1995), pp. 244–258.

[3] D. Cheriton and K. Harty, *Application-controlled physical memory using external page-cache management*, Technical report draft, Department of Computer Science, Stanford University, Stanford, CA, 1991.

[4] P. J. Denning, *Working sets past and present*, IEEE Trans. Software Engrg., SE-6 (1980), pp. 64–84.

[5] A. Fiat, R. Karp, M. Luby, L. McGeoch, D. Sleator, and N. Young, *On competitive algorithms for paging problems*, J. Algorithms, 12 (1991), pp. 685–699.

[6] P. A. Franaszek and T. J. Wagner, *Some distribution-free aspects of paging performance*, J. Assoc. Comput. Mach., 21 (1974), pp. 31–39.

[7] A. R. Karlin, M. S. Manasse, L. Rudolph, and D. D. Sleator, *Competitive snoopy caching*, Algorithmica, 3 (1988), pp. 70–119.

[8] T. Kilburn, D. B. G. Edwards, M. J. Lanigan, and F. H. Sumner, *One-level storage system*, IRE Trans. Elect. Computers, 37 (1962), pp. 223–235.

[9] M. S. Manasse, L. A. McGeoch, and D. D. Sleator, *Competitive algorithms for on-line problems*, J. Algorithms, 11 (1990), pp. 208–230.

[10] D. McNamee and K. Armstrong, *Extending the Mach external pager interface to accommodate user-level page replacement policies*, Technical report 90-09-05, Department of Computer Science and Engineering, University of Washington, Seattle, WA, 1990.

[11] G. S. Shedler and C. Tung, *Locality in page reference strings*, SIAM J. Comput., 1 (1972), pp. 218–241.

[12] D. D. Sleator and R. E. Tarjan, *Amortized efficiency of list update and paging rules*, Comm. Assoc. Comput. Mach., 28 (1985), pp. 202–208.

[13] J. R. Spirn, *Program Behavior: Models and Measurements*, Elsevier Computer Science Library. Elsevier, Amsterdam, 1977.

SIAM J. COMPUT.
Vol. 25, No. 3, pp. 498–519, June 1996

ON THE VALUE OF COORDINATION IN DISTRIBUTED DECISION MAKING*

SANDY IRANI† AND YUVAL RABANI‡

Abstract. We discuss settings where several "agents" combine efforts to solve problems. This is a well-known setting in distributed artificial intelligence. Our work addresses theoretical questions in this model which are motivated by the work of Deng and Papadimitriou [*Proc.* 12*th IFIPS Congress*, Madrid, 1992; *Proc. World Economic Congress*, Moscow, 1992]. We consider optimization problems, in particular load balancing and virtual circuit routing, in which the input is divided among the agents. An underlying directed graph, whose nodes are the agents, defines the constraints on the information each agent may have about the portion of the input held by other agents. The questions we discuss are as follows: Given a bound on the maximum out-degree in this graph, which is the best graph? What is the quality of the solution obtained as a function of the maximum out-degree?

Key words. analysis of algorithms, distributed computation, competitive analysis, load balancing, virtual circuit routing

AMS subject classifications. 68Q22, 68Q25

1. Introduction.

In recent years, there has been a great deal of research activity focused on analyzing algorithms which must compute using partial information about the problem to be solved. Much of this research effort has focused on *on-line* algorithms, where the limitation is due to temporal constraints: the input is arriving a piece at a time, and output must be produced before all the input arrives. The study of on-line algorithms is motivated by the fact that many problems which arise in a wide variety of settings are inherently on-line (i.e., one doesn't have the luxury of being able to collect all the information about an the instance of the problem before a partial answer must be produced). However, part of the reason for the recent interest in this area is the introduction of an appealing means of evaluating on-line algorithms called *competitive analysis* [18, 15]. The idea is to determine the quality of an on-line algorithm by comparing its performance to the performance of the optimal algorithm that can see the entire input in advance. Thus we measure what is lost by solving the problem on-line.

The work in this paper follows a model proposed by Deng and Papadimitriou [11] and further discussed by Papadimitriou and Yannakakis [17] who extend the use of competitive analysis to a more general setting than on-line algorithms. They discuss settings where several "agents" combine efforts to solve problems. The idea is that global information about a problem to be solved may be lacking due to spatial (or other) constraints. The framework suggested in [11] is to model specific constraints in the availability of information and study the solution quality that can be obtained under these information regimes. In [17], the information regime is determined by the input. Linear programming is considered, where each agent is responsible for a

* Received by the editors January 12, 1994; accepted for publication (in revised form) November 1, 1994. An earlier version of this paper appeared in *Proc.* 34*th IEEE Symposium on Foundations of Computer Science*, Palo Alto, 1993, under the title "On the value of information in coordination games."

† Department of Information and Computer Science, University of California, Irvine, CA 92717 (irani@ics.uci.edu). The research of this author was supported in part by NSF grant CCR-9309456.

‡ Laboratory for Computer Science, Massachusetts Institute of Technology, 545 Technology Square, Cambridge, MA 02139 (rabani@theory.lcs.mit.edu). The research of this author was performed at the International Computer Science Institute, Berkeley, CA and supported by a Rothschild post-doctoral fellowship.

variable or a set of variables and sees all constraints involving those variables.

We take a different approach that is suitable for the case where information is available at a price. Rather than analyzing particular constraint structures, we focus on the best constraint structure given a bound on the amount each agent communicates. We address the following questions. To what extent is it useful for the agents to communicate information about the piece of the input that each holds? What is the most effective pattern of communication? If some a priori information about the problem instance is known, how can this information be used to improve the solution quality?

1.1. The model. We consider optimization problems in the following context: A set of agents $A = \{a_0, a_1, \ldots, a_{n-1}\}$ are given an instance I of an optimization problem. Each agent is presented with a portion of the input: a_i gets input I_i, where $I = \cup_i I_i$. We assume that the objective function is not part of the input and is known to all agents.

A *strategy* S for the agents is a pair (G, D), where G is a directed graph and D is a set of algorithms, one for each agent. We refer to G as the *knowledge graph* because it represents the information available to each agent: a directed edge (a, b) in G means that agent a knows of the input portion given to agent b. Each agent's decisions are a function only of the input it knows: a set of algorithms D is *valid* for a knowledge graph G if the algorithm for each agent a_j is a function only of I_j and all I_k such that $(j, k) \in E$. If $S = (G, D)$ is a strategy, then D must be valid for G. In some cases, we refer to an undirected knowledge graph in which case an edge between agents a and b indicates that a and b know each other's input.

We denote by $\text{cost}_S(I)$ the value of the objective function for a strategy S on input I, and we denote by $\text{cost}(I)$ the value of the objective function for the optimal, global strategy on input I. A strategy S is *c-competitive* if for every I, $\text{cost}_S(I) - c \cdot \text{cost}(I)$ is bounded by a constant. The *competitive ratio* of S, denoted c_S, is the infimum over all c such that S is c-competitive.

If the knowledge graph is predetermined, then the goal is to devise the best algorithm with the given limitation in information. Thus, for a given knowledge graph G, we would like to choose the best set of algorithms D that are valid for that graph. The competitive ratio for a knowledge graph, denoted c_G, is the infimum of $c_{(G,D)}$ over algorithms D that are valid for G. On the other hand, if complete information is available (that is, any two agents can communicate) but at a cost, we would like to determine to what extent the solution can be improved with more information. Thus, we consider strategies that are constrained by limiting the maximum degree of a vertex in G. If G has maximum degree r, we call S an *r-strategy*. c_r is the infimum of c_S over all r-strategies. What pattern of communication and what algorithm are best for a given limitation in communication: for a given r, what r-strategy achieves c_r? To what extent is communication useful: how does c_r decrease as r increases? We study these general issues with respect to two specific problems: load balancing and virtual circuit routing.

Awerbuch et al. [3] study a problem very similar in flavor to the problems we consider here. They consider the number of steps necessary to broadcast a message in a fixed network. They show bounds on the number of steps necessary as a function of the radius of the network graph each vertex knows. Unlike our model, the processors do not get to choose the information they acquire. However, for the problems we consider, it is always optimal for the agents to communicate by grouping themselves into disjoint cliques and completely sharing information within a clique.

This model has several applications. We mention some of them here.

Parallel programming. It is now clear that realistic models of parallel computation must address communication overhead as well as processing time [1, 10, 12, 16, 19]. It seems easier to design and implement a parallel program where the parallel tasks are oblivious to each other as much as possible. Coordination between parallel tasks requires additional programming and increases communication overhead. Our work speaks to the tradeoff between the amount of coordination in a parallel program and the effectiveness of that program in solving specific problems. Particularly, the design of system services such as batch execution might benefit from this analysis.

High-speed network management. High-speed networks are expected to serve a large number of users bidding for a variety of services. The allocation of network resources by a centralized network manager becomes impractical under such conditions. As pointed out in [17], the multiple-agents model is suitable for discussing performance degradation due to the distributed nature of network management. Specifically, our results relate to the following questions: Network managers are requested to allocate virtual connecting paths between pairs of sites. Each virtual circuit consumes a fixed bandwidth. What is the required capacity of network links and switches to handle the expected traffic? How does this capacity change as a function of communication among network managers? What network structures support distributed management?

Large-scale planning. The question of cooperation among communicating problem solvers is considered fundamental in distributed AI, as it addresses planning problems in a large-scale system or organization that is faced with a rapidly changing environment. See [6] for a comprehensive collection of papers in the field. The AI approach tends to be either qualitative or experimental. Recent theoretical results [11, 17] as well as this work focus on quantitative analysis of such problems.

1.2. Outline of results. We consider the load-balancing problem discussed in [11]. Each agent gets a set of jobs to be executed, where the length of each job is known in advance. The agents redistribute the jobs among themselves. Their goal is to minimize the maximum load on an agent. The optimal strategy clearly divides the jobs evenly among all the agents, or as evenly as possible given the granularity of the job lengths.

Deng and Papadimitriou give a complete analysis of the problem of three agents scheduling jobs on two processors for all possible knowledge graphs. They also show that for an arbitrary number n of agents distributing jobs among themselves, when the agents do not communicate at all (i.e., the knowledge graph has no edges), then there is a way for each agent to redistribute its jobs that achieves a competitive ratio of $2\sqrt{n}$. Furthermore, for any deterministic strategy, there is a way of assigning n jobs of length 1 to the n agents such that some agent receives at least $\sqrt{n}$ jobs. Thus, if G has no edges, then $\sqrt{n} \leq c_G \leq 2\sqrt{n}$.

We generalize their results to show that for a fixed knowledge graph G, $\sqrt{\alpha(G)} \leq c_G \leq 2\sqrt{\phi(G)}$, where $\alpha(G)$ is the size of the maximum independent set of G and $\phi(G)$ is the size of the minimum clique cover in G. Thus, by choosing G to be a collection of disjoint $(r+1)$-cliques, $\sqrt{n/(r+1)} \leq c_r \leq 2\sqrt{n/(r+1)}$. The factor of two can be reduced when all the job lengths are identical.

As one might suspect, randomization is a very powerful tool in this setting. Deng and Papadimitriou show for the empty knowledge graph that if each agent sends each job to a random destination, then the competitive ratio is $\log n / \log\log n$. We show an asymptotically matching lower bound (also shown independently by Alon

[2]). Furthermore, we consider r-strategies for all r. We show tight bounds of $c_r \in \Theta(\log(n/r)/\log\log(n/r))$. The lower bound holds for any distribution of r-regular graphs, i.e., even when global information available to all agents is hidden from the adversary. For example, the lower bound holds even if the agents can organize themselves into random collections of disjoint $(r+1)$-cliques. The upper bound follows, as in the deterministic case, from an algorithm that gives $c_G = O\left(\log\phi(G)/\log\log\phi(G)\right)$ for all knowledge graphs G. The upper bound requires that agents which share their pieces of input can also toss common coins (but no global coins are needed).

We also consider questions that relate to virtual circuit routing. The problem is to route permutations in a network where each agent is responsible for selecting the route of a single input. The goal is to minimize the node or edge congestion. When no information is available to the agents besides their own destination assignment, this is the well-studied question of *oblivious* routing. If the paths for each input–output pair are precomputed by some central algorithm, we have the problem of *global* routing—also a very well-understood problem. We consider routing where the available information is in the spectrum between the oblivious and the global cases. Each agent (which represents a single input vertex in the network) knows its own destination and the destinations of some of the other agents and must decide on its path using the available information. The knowledge graph represents what information is available to which agents, and we determine the benefit obtained when each agent has degree at most r in the knowledge graph.

We consider N-node, degree-d networks with n input and n output nodes. We assume that the network is optimal for the required task, i.e., the agents may choose the structure of the network. For that reason, we do not give a competitive analysis but rather a worst-case analysis. We show an r-strategy for routing in a $\log n$-dimensional Beneš network [4, 20] with maximum edge congestion of $\sqrt{n/r}$. The lower bounds on oblivious routing of [7, 14] can be modified to show lower bounds of $(1/2d)\sqrt{n^2/(Nr)}$ on edge congestion and $(1/2)\sqrt{n^2/(Nr(d+1))} - (n/2Nr)$ on node congestion. We show a lower bound on node congestion of $\Omega\left(\log(n/r)/\log\log(n/r)\right)$ for randomized r-strategies. The lower bound follows the lower bound on oblivious single-port routing of Borodin et al. [9]. All of our bounds match the previously known bounds for $r = 1$ (oblivious routing) and $r = n$ (global routing).

2. Load balancing.

Each agent decides deterministically where to send its jobs based on the set of jobs it has and the set of jobs each of its neighbors in the knowledge graph has. Let $\phi(G)$ be the size of the minimum clique cover for a graph G and $\alpha(G)$ the size of the maximum independent set of G. We prove the following theorem.

THEOREM 1. *For every graph G, $\sqrt{\alpha(G)} \le c_G \le 2\sqrt{\phi(G)}$.*

Proof. Let S be an independent set. Let $|S| = s$. Give $n/\sqrt{s}$ jobs to each agent in S. After the jobs have been redistributed, some agent a will get $\sqrt{s}$ jobs. Pick a subset of at most $\sqrt{s}$ agents such that the total number of jobs given to agent a by agents in the subset is at least $\sqrt{s}$. Now instead, give $n/\sqrt{s}$ jobs only to agents in the subset. Since there are at most n jobs total, the optimal cost is one. Agent a still gets $\sqrt{s}$ jobs.

Let $\phi(G) = \Phi$. Partition the vertices into Φ cliques, $c_0, \ldots, c_{\Phi-1}$. A clique acts as one agent since all the agents know the inputs of the other agents in the clique. Fix an arbitrary ordering on the agents: $a_0, \ldots, a_{n-1}$. Let $m = n/\Phi$. Each clique c_i divides its tasks into at most n groups so as to minimize the maximum length of a group's tasks. Then it sorts the groups in decreasing order by length. Clique c_i sends the jobs in group k to agent $a_{\lfloor mi \rfloor + k \pmod{n}}$.

Fix some agent a_j. Let i_j be the number that satisfies $\lfloor m i_j \rfloor \leq j < \lfloor m(i_j+1) \rfloor$. Let t_i denote the number of jobs given to agent j from the clique $c_{i_j - i \pmod{\Phi}}$ (the clique that precedes clique i_j in the ordering by i). Fix some k such that $1 \leq k < \Phi$. We divide the work sent to agent a_j into two parts: $T_1 = \sum_{i=0}^{k-1} t_i$ and $T_2 = \sum_{i=k}^{\Phi-1} t_i$. The total job length sent to agent a_j is $T = T_1 + T_2 \leq 2\max\{T_1, T_2\}$. We can lower bound the cost to the optimal strategy by $\max_{0 \leq i < k} t_i$ because each clique divides its jobs into groups so that the maximum total length in any group is minimized. If $T_1 \geq T_2$,

$$\frac{T}{\max_{0 \leq i < k} t_i} \leq \frac{2T_1}{\max_{0 \leq i < k} t_i} \leq 2k.$$

Clique $c_{i_j - i \pmod{\Phi}}$ sends at least t_i jobs to agents $\lfloor (i_j - i)m \pmod{n} \rfloor, \ldots, \lfloor m i_j \rfloor$. Thus the total number of jobs originating at clique $c_{i_j - i \pmod{\Phi}}$ is at least $(\lfloor m i_j \rfloor - \lfloor m(i_j - i) \pmod{n} \rfloor + 1)t_i \geq m i t_i$. The total length of all the jobs in the system is at least $\sum_{i=k}^{\Phi-1} m i t_i \geq mk \sum_{i=k}^{\Phi-1} t_i$. Thus, the optimal solution sends jobs of total length at least

$$\frac{mk \sum_{i=k}^{\Phi-1} t_i}{n} = \frac{\sum_{i=k}^{\Phi-1} t_i}{\Phi/k} = \frac{kT_2}{\Phi}$$

to some agent. Thus, if $T_2 \geq T_1$, then

$$\frac{T}{kT_2/\Phi} \leq \frac{2T_2}{kT_2/\Phi} = 2\Phi/k.$$

Picking k to minimize the maximum of the two bounds $2\Phi/k$ and $2k$, we get $k = \sqrt{\Phi}$ which yields an upper bound of $2\sqrt{\Phi}$. □

COROLLARY 2. $\sqrt{n/(r+1)} \leq c_r \leq 2\sqrt{n/(r+1)}$.

Proof. The lower bound follows from the fact that any graph with maximum degree r has an independent set of size at least $n/(r+1)$ (the greedy algorithm finds such a set). For the upper bound, partition the set of agents into disjoint subsets of size $r+1$ each. The knowledge graph consists of $n/(r+1)$ complete graphs, one on each of the subsets. □

Remark. The upper bound can be improved to $\sqrt{2\Phi}$ when all the job lengths are the same. When the jobs lengths are not the same, it is an NP-hard problem to divide a set of jobs into at most n groups so as to minimize the maximum total length in any group. However, there is a polynomial-time approximation algorithm which comes within a factor of 4/3 of the optimal solution [13]. Thus, the above upper bound can be achieved by polynomial-time bounded agents with an extra factor of 4/3 in the ratio.

A natural question to ask is whether the upper or lower bound of Theorem 1 is tight. In order to answer this question, we need to examine a class of graphs whose maximum independent set and minimum clique cover differ. The distribution $\mathcal{G}(n, 1/2)$ is the distribution over all n-node undirected graphs where each pair of nodes is adjacent independently with probability 1/2. If G is drawn according to $\mathcal{G}(n, 1/2)$, then with high probability, $\phi(G) = \Omega(n/\log n)$ and $\alpha(G) = O(\log n)$ [5]. These facts combined with the following claim imply that the upper bound of Theorem 1 is not tight for the case when all job lengths are 1. We suspect that the lower bound is not tight either, that there are graphs G for which c_G lies strictly between $\sqrt{\alpha(G)}$ and $\sqrt{\phi(G)}$.

CLAIM. *If G is drawn at random from $\mathcal{G}(n, 1/2)$, then with high probability, $c_G = O(n^{1/3} \log n)$ when all job lengths are uniform.*

Proof. We can assume that no agent receives more than n jobs. If an agent receives x jobs, it can distribute $n\lfloor x/n \rfloor$ jobs evenly among the agents and it is left with the problem of distributing the remaining $x \bmod n$ jobs.

An *i-adversary* gives each agent either 0 jobs or x jobs, where $2^i \leq x < 2^{i+1}$. Suppose that for every $0 \leq i \leq \log n$, we can achieve a ratio of c against an i-adversary. Then we can achieve a ratio of $c(\log n + 1)$ against any adversary. Let the *i-agents* be those agents that start with y jobs such that $2^i \leq y < 2^{i+1}$. Each i-agent follows the strategy against the i-adversary with the following change. For every non-i-agent whose input it can "see," it assumes that that agent received 0 jobs. Let M_i be the number of jobs given to the agent who receives the maximum number of jobs from i-agents after the jobs have been redistributed. Let X_i be the total number of jobs given to i-agents by the adversary. Let $t = \log n$. The agent with the maximum number of jobs after redistribution has no more than $\sum_{i=0}^t M_i$ jobs. The optimal solution gives at least $\lceil (\sum_{i=0}^t X_i)/n \rceil$ jobs to some agent. We are guaranteed that $M_i / \lceil X_i/n \rceil \leq c$.

Since $\sum_{i=0}^t \lceil (X_i/n) \rceil \leq (t+1) \lceil (\sum_{i=0}^t X_i)/n \rceil$, we have that

$$\frac{\sum_{i=0}^t M_i}{\lceil \sum_{i=0}^t \frac{X_i}{n} \rceil} \leq \frac{(t+1)\sum_{i=0}^t M_i}{\sum_{i=0}^t \lceil \frac{X_i}{n} \rceil}$$

$$\leq (t+1) \cdot \max_i \left\{ \frac{M_i}{\lceil \frac{X_i}{n} \rceil} \right\} \leq c \cdot (\log n + 1).$$

Now for every $0 \leq i \leq \log n$, we show a strategy against an i-adversary. Then we show that with high probability, G has a certain property which ensures that each strategy achieves a ratio of $O(n^{1/3})$. Say that G has property A if for every subset of $8n^{1/3}$ nodes in G, the subgraph induced by that subset has more than $4n^{2/3}$ edges. We have the following lemma.

LEMMA 3. *When G is chosen according to $\mathcal{G}(n, 1/2)$, then*

$$\Pr[G \textit{ does not have property } A] \leq 2^{n^{1/3}(8 \log n - \frac{5}{3} n^{1/3})}.$$

Proof. Let $s = n^{1/3}$. Consider a fixed subset S of $8s$ vertices. What is the probability that there are at most $4s^2$ edges in the subgraph induced by S? There are at least $\binom{8s}{2} \geq 28s^2$ edge slots. Thus the expected number of edges is at least $14s^2$. Using Chernoff bounds, the probability that a fixed subset of $8s$ vertices induces a subgraph with no more than $4s^2$ edges is at most

$$e^{-(10s^2)^2/2 \cdot 28s^2} \leq 2^{-\frac{5}{3}s^2}.$$

Thus, the probability that there is a subset of size $8s$ vertices in the graph whose induced subgraph has fewer than $4s^2$ edges is at most

$$\binom{n}{8s} 2^{-\frac{5}{3}s^2} \leq n^{8n^{1/3}} 2^{-\frac{5}{3}n^{2/3}} \leq 2^{n^{1/3}(8 \log n - \frac{5}{3} n^{1/3})}. \qquad \square$$

Now we will show the strategy against an i-adversary. Our strategy gives the desired ratio if G has property A. Therefore, if property A holds, the strategies for all i give the desired ratio. Let $m = 2^{\lceil \log n \rceil}$. There will be n agents sending jobs and *m receivers.* The jobs sent to receiver j are sent to agent $j \pmod{n}$. An agent gets no more than twice the load of the most heavily loaded receiver. We will determine the ratio of the most heavily loaded receiver in the distributed solution to the most heavily loaded agent in the optimal solution. Let $X = 2^i$. Let x_j be the number of jobs that originate with agent j. Since we are playing against an i-adversary, $x_j = 0$ or $X \le x_j < 2X$ for all j.

If $X \ge n^{2/3}$, then agent j sends a job to receiver $k + j \bmod n$ for all $1 \le k \le x_j$. If some agent gets a jobs after redistribution, then there are at least aX jobs among all the agents. The optimal solution gives at least aX/n to some agent. This gives a ratio of at most n/X.

If $X \le n^{1/3}$, then each agent keeps all the jobs that it receives.

If $n^{1/3} \le X \le n^{2/3}$, then divide the receivers into m/X groups of X consecutive receivers. The agents are also divided into consecutive groups of X. There are only $\lceil n/X \rceil$ such groups and the last group may have fewer than X agents.

There are two cases.

Case 1. If an agent in group i gets jobs and is adjacent to no more than $n^{1/3}$ agents with jobs in its group, then it distributes its jobs evenly among the receivers in group i. In this case, an agent gives at most two jobs to each receiver.

Case 2. If an agent j is adjacent to at least $n^{1/3}$ agents with jobs in its group, it sends its jobs to receiver $(Xk + (j \bmod X)) \pmod{m}$ for $0 \le k < x_j$. In other words, each agent can be specified by the group to which it belongs (j div X) and its number within the group ($j \pmod{X}$). Starting with group 0 and cycling through the m/X groups of receivers, agent j gives its jobs to the receiver in each group which has the same number within its group.

Now consider a group of X agents with more than $8n^{1/3}$ agents that fall into Case 1. Pick $8n^{1/3}$ of them and call this set S. Every agent in S has fewer that $n^{1/3}$ edges to agents in S. That means that the subgraph induced by S has no more than $4n^{2/3}$ edges (since the sum of degrees is twice the number of edges); i.e., G does not have property A. Thus, by Lemma 3, with high probability G has the property that it is impossible to have more than $8n^{1/3}$ agents that fall into Case 1. For the remainder of the proof, we assume that this is the case. Using the fact that an i-agent from Case 1 gives at most two jobs to each agent in its group, we can conclude that the number of jobs given to an agent by i-agents from Case 1 is at most than $16n^{1/3}$.

So suppose that some receiver j gets l jobs from Case 2 agents. Receiver j only gets these jobs from agents whose number is congruent to $j \bmod X$. Each of these agents is in a different group and spreads its jobs as evenly as possible among the groups of receivers. Since each agent starts with at most $2X$ jobs and there are m/X groups, at most $\lceil 2X^2/m \rceil \le \max\{1, 4X^2/m\}$ jobs that end up with agent j come from the same group. Thus, there are at least $\min\{l, lm/4X^2\}$ groups where $n^{1/3}$ of the agents get jobs. Each such group has at least $Xn^{1/3}$ jobs. Therefore, the total number of jobs is at least $Xn^{1/3} \min\{l, lm/4X^2\}$, and the optimal solution gives at least $(X/n^{2/3}) \min\{l, lm/4X^2\}$ to some agent. Thus, the competitive ratio due to Case 2 is at most $\max\{n^{2/3}/X, 4X/n^{1/3}\}$. For $n^{1/3} \le X \le n^{2/3}$, the ratio is at most $4n^{1/3}$. □

We show the following upper bound on randomized strategies.

THEOREM 4. *For every knowledge graph G, there is a randomized load-balancing strategy whose competitive ratio is in $O(\log(\Phi)/\log\log(\Phi))$, where $\Phi = \phi(G)$.*

Proof. Let $m = \lceil n/\Phi \rceil$. Pick a clique cover of G of size Φ. We assume that the agents within a clique know the set of jobs assigned to the other agents in the clique as well as their random bits. Thus, each clique operates as a single agent. The agents will be divided into Φ sets of size m. Each agent will be indexed by a pair (i, j), where $0 \leq i \leq \Phi - 1$ and $0 \leq j \leq m - 1$. i indicates the set to which the agent belongs and j indicates the place within the set.

Each clique c_l divides its tasks into at most n sets such that the maximum length of a set's tasks is minimized. Then it sorts the sets in decreasing order of length. Each set will be indexed by a triplet (i, j, l) where $0 \leq i, l \leq \Phi - 1$ and $0 \leq j \leq m - 1$. The third index denotes the clique where the jobs originate. The first two indices identify the specific set at clique l. Let $s(i, j, l)$ be the total length of the jobs in set (i, j, l). They are sorted so that $s(i, j, l) \geq s(i', j', l)$ if $i < i'$ or if $i = i'$ and $j \leq j'$. For each $i \in \{0, \ldots, \Phi - 1\}$, clique c_l draws $k \in \{0, \ldots, \Phi - 1\}$ uniformly at random. Then for all $j \in \{0, \ldots, m - 1\}$, it sends all the jobs in set (i, j, l) to agent (k, j).

Denote the optimal cost for this instance by OPT. Denote the cost due to the distributed algorithm by the random variable D. We wish to show that $E[D] \in O((\log \Phi / \log \log \Phi)\text{OPT})$.

Consider a related problem: we have Φ agents. Agent l gets a set of Φ jobs of lengths $s(0, 0, l)$, $s(1, 0, l), \ldots, s(\Phi - 1, 0, l)$. (Some of these could be of length 0.) Let OPT' be the cost when the jobs are optimally distributed among the Φ agents. Let D' be the random variable denoting the cost of the distributed algorithm which sends each job to a random agent. The result of [11] gives that $E[D'] \in O((\log \Phi / \log \log \Phi)$ $\text{OPT}')$. Clearly, D and D' are identically distributed, so $E[D] = E[D']$. We show that $\text{OPT} \geq (1/3)\text{OPT}'$, which completes the proof.

Divide the jobs from the second problem into two sets:

$$T_1 = \sum_{l=0}^{\Phi-1} s(0, 0, l),$$

$$T_2 = \sum_{i=1}^{\Phi-1} \sum_{l=0}^{\Phi-1} s(i, 0, l).$$

Let $\text{MAX} = \max_{0 \leq l \leq \Phi-1} s(0, 0, l)$ Clearly, $\text{MAX} \geq T_1/\Phi$. We claim two facts.

1. $\text{OPT} \geq \max\{\text{MAX}, T_2/\Phi\}$;
2. $\text{OPT}' \leq \text{MAX} + (T_1 + T_2)/\Phi$.

The theorem follows from the two claims because

$$\text{OPT}' \leq 2\text{MAX} + \frac{T_2}{\Phi} \leq 3\text{OPT}.$$

To prove the first claim, observe that in the first problem each clique divides the jobs into sets so as to minimize the maximum length of the jobs in any set. So the length of the jobs in any set is a lower bound on the optimal solution. The second part of the "max" in claim 1 follows from

$$\text{OPT} \geq \frac{1}{n} \sum_{j=0}^{m-1} \sum_{i=0}^{\Phi-1} \sum_{l=0}^{\Phi-1} s(i, j, l).$$

Since for all $1 \leq i < m$, $\sum_{j=0}^{m-1} s(i, j, l) \geq m \cdot s(i + 1, 0, l)$,

$$\frac{1}{n} \sum_{j=0}^{m-1} \sum_{i=0}^{\Phi-1} \sum_{l=0}^{\Phi-1} s(i, j, l) \geq \frac{m}{n} \sum_{i=1}^{\Phi-1} \sum_{l=0}^{\Phi-1} s(i, 0, l)$$

$$\geq \frac{mT_2}{n} = \frac{T_2}{\Phi}.$$

To see the second claim, observe the discrepancy disc in the optimal solution. The discrepancy is the difference between the maximum load on an agent and the minimum load on an agent. Clearly, disc $\leq$ MAX. Since $(T_1 + T_2 + \Phi \cdot \text{disc})/\Phi$ is an upper bound on the optimal cost, the claim follows. □

COROLLARY 5. *There are randomized load balancing r-strategies for all n and r with a competitive ratio in $O(\log(n/r)/\log\log(n/r))$.*

Proof. Partition the set of agents into disjoint subsets of size $r+1$ each. The knowledge graph consists of $\lceil n/(r+1)\rceil$ complete graphs, one on each of the subsets. □

Corollary 5 is tight up to a constant factor, as the following theorem shows.

THEOREM 6. *If $\sqrt[4]{r(n)/n} \longrightarrow 0$ as $n \to \infty$, then for every sufficiently large n and for $r = r(n)$, the competitive ratio of every randomized load-balancing r-strategy on n agents is in $\Omega(\log(n/r)/\log\log(n/r))$.*

Remark. If $\sqrt[4]{r(n)/n} \geq \epsilon > 0$ for all n, then $(n/r(n)) \leq \epsilon^{-4}$. So, we get a lower bound of $\Omega(\log(n/r)/\log\log(n/r))$ for all $r < n$. Alon [2] has independently shown a lower bound of $\Omega(\log n/\log\log n)$ for the special case of an empty knowledge graph. This bound follows from our proof as well.

Proof. We need to consider only r-regular knowledge graphs (i.e., with *out-degree* r). Applying von Neumann's minimax principle (see [21, 8]), we show a probability distribution over inputs which beats every deterministic algorithm.

For the sake of completeness, we state and prove the exact claim that is needed (see [8, Lem. 7.2] for the original version).

LEMMA 7. *Let $\tilde{I}$ be a probability distribution over a finite sample space of inputs such that for every deterministic r-strategy S, $E_{\tilde{I}}[\text{cost}_S(\tilde{I}) - c\cdot\text{cost}(\tilde{I})] \geq 0$. Then for every randomized r-strategy $\tilde{S}$, there exists an input instance $I = I(\tilde{S})$ for which*

$$\frac{E_{\tilde{S}}[\text{cost}_{\tilde{S}}(I)]}{\text{cost}(I)} \geq c.$$

Proof. Fix $\tilde{S}$. $\tilde{S}$ is a probability distribution over deterministic strategies S_χ. Since for every χ, $E_{\tilde{I}}[\text{cost}_{S_\chi}(\tilde{I}) - c\cdot\text{cost}(\tilde{I})] \geq 0$, we have that $E_\chi[E_{\tilde{I}}[\text{cost}_{S_\chi}(\tilde{I}) - c\cdot\text{cost}(\tilde{I})]] \geq 0$. We may switch the order of integration since all expectations are finite. We get $E_{\tilde{I}}[E_\chi[\text{cost}_{S_\chi}(\tilde{I}) - c\cdot\text{cost}(\tilde{I})]] \geq 0$, or $E_{\tilde{I}}[\text{cost}_{\tilde{S}}(\tilde{I}) - c\cdot\text{cost}(\tilde{I})] \geq 0$. Therefore, there exists $I \in \tilde{I}$ for which $\text{cost}_{\tilde{S}}(I) - c\cdot\text{cost}(I) \geq 0$. □

The distribution we choose gives every agent d jobs independently with probability p and 0 jobs otherwise. We use $d = \sqrt[4]{r^3 n}$ and $(4/d) < p < (4e/d)$ so that $p(1-p)^r d = 4$. Notice that $d \in \omega(r) \cap o(n)$. The expected cost of the optimal algorithm is at most $8e$. Our goal is to show that the expected cost of any deterministic r-strategy is in $\Omega(\beta)$, where $\beta = \log(n/r)/\log\log(n/r)$.

Let $k = \log(d/r)/\log\log(d/r)$. By our choice of d, $k \geq \beta/4$. We need the following lemma.

LEMMA 8. *If $r(n) \in o(d(n))$ and if $Z = Z(n)$ is distributed according to the binomial distribution $B(4(r+1)/d, d/4(r+1))$, then for n large enough,*

$$\text{Prob}[Z \geq k] \geq \frac{r}{d}.$$

Proof. Let $\lambda = E[Z] = 1$. $\text{Prob}[Z = k]$ (which is a lower bound for $\text{Prob}[Z \geq k]$) can be estimated using the Poisson approximation to the binomial distribution. It gives

$$\mathrm{Prob}[Z=k] > p(k;\lambda)e^{-\frac{k^2}{d/4(r+1)-k}-\frac{\lambda^2}{d/4(r+1)-\lambda}},$$

where $p(k;\lambda)=e^{-\lambda}\lambda^k/k!$.

We can estimate $p(k;\lambda)$ for large k using Stirling's formula as follows:

$$p(k,1)=\frac{1}{e(k)!}\geq\frac{a}{\sqrt{k}}\left(\frac{e}{k}\right)^k$$

for some constant a.

Also, $e^{-\frac{k^2}{d/4(r+1)-k}-\frac{\lambda^2}{d/4(r+1)-\lambda}} \longrightarrow 1$ as $n\to\infty$, so for sufficiently large n this is lower bounded by $1/2$.

We have, for sufficiently large n,

$$\log\left(\frac{2\sqrt{k}}{a}\left(\frac{k}{e}\right)^k\right)\leq k\log k\leq\log\left(\frac{d}{r}\right). \quad \square$$

The condition that $\sqrt[4]{r(n)/n}\longrightarrow 0$ as $n\to\infty$, implies that $r(n)\in o(d(n))$.

We use Lemma 8 to bound a similar distribution. A vertex of G is said to be *chosen* if it gets jobs. A vertex is *isolated* if it is chosen and none of its neighbors are chosen.

LEMMA 9. *Let A be a subset of the vertices of G. To each vertex $v\in A$, assign an integer weight W_v between 1 and $k/8$ such that $\sum_{v\in A}W_v\geq d/4$. Let Y be the sum of the weights of the isolated vertices in A. Let Z be as in Lemma 8. Then for n sufficiently large,*

$$\mathrm{Prob}\left[Y\geq\frac{k}{2}\right]\geq\frac{1}{e^9}\mathrm{Prob}[Z\geq k].$$

Proof. Recall that $k=\log(d/r)/\log\log(d/r)$. Let $s=d/4$. We may assume that $\sum_{v\in A}W_v=s$; otherwise, we reduce weights and remove 0-weighted vertices from A until equality holds. For each vertex $v\in A$, we introduce W_v indicator variables. Each variable is set to 1 if its corresponding vertex is isolated and to 0 otherwise. Denote the indicator variables by $Y_1,\ldots,Y_s$. $Y=\sum Y_j$. For X a k-subset of $\{1,\ldots,s\}$, $V_Y(X)$ is the subset of vertices associated with $\{Y_i \mid i\in X\}$.

The proof proceeds in two steps. First, we relate the probability of the desired event in G to the probability of a similar event in a graph G_U, derived from G, where vertex weights are 1. Then we relate the probability of the desired event in G_U to the distribution of Z.

We form a graph G_U which is identical to G except that for every vertex $v\in A$, we have a set S_v of W_v nodes in G_U. If v and u are adjacent in G, then in G_U every vertex in S_u is adjacent to every vertex in S_v. Note that the degree in G_U is at most $kr/8$. Let $A_U=\cup_{v\in A}S_v$. Notice that $|A_U|=s$. Let U_i denote the indicator variable that is 1 if vertex i in A_U is isolated and 0 otherwise when each vertex of G_U is chosen independently of the others with probability p. Let $U=\sum U_j$.

Now consider a particular graph G_Z of s vertices consisting of disjoint $(r+1)$-cliques. Let $Z_1,\ldots,Z_s$ be the indicator variables which indicate for each vertex whether it is isolated when each vertex is chosen with probability p. Clearly, the distributions of Z from Lemma 8 and $\sum Z_j$ are identical.

We wish to show the following:

1. Prob$[Y \geq k/2] \geq$ Prob$[U \geq k]/e^4$.
2. Prob$[U \geq k] \geq$ Prob$[Z \geq k]/e^5$.

Proof of part 2. Define

$$S_U = E\left[\sum_{X \subset \{1,\ldots,s\}, |X|=k} \prod_{j \in X} U_j\right].$$

Similarly,

$$S_Z = E\left[\sum_{X \subset \{1,\ldots,s\}, |X|=k} \prod_{j \in X} Z_j\right].$$

Clearly, S_U is an upper bound on Prob$[U \geq k]$ and S_Z is an upper bound on Prob$[Z \geq k]$. First we prove that $S_U \geq S_Z/e^2$.

Examine a fixed k-set X of $\{1,\ldots,s\}$. When does $\prod_{j \in X} U_j = 1$? This happens when X forms an independent set, all the vertices in X are chosen, and all the vertices in the neighborhood set of X are not chosen. Notice that $|X| = k$ and the neighborhood set of X has at most $rk^2/8$ vertices. Thus if X forms an independent set, then $\prod_{j \in X} U_j = 1$ with probability at least $p^k(1-p)^{rk^2} \geq p^k/e \geq p^k(1-p)^{rk}/e$. (The first inequality follows from the fact that for sufficiently large n, $rk^2 \ll 1/p$. Since $(1-1/x)^{x-1} > e^{-1}$ for all $x \geq 2$, the inequality follows.)

Now let's look at the Z_j's. If X forms an independent set, then the neighborhood set of X has size rk (because the graph is composed of disjoint $(r+1)$-cliques). Thus, if X is an independent set, then $\prod_{j \in X} Z_j = 1$ with probability exactly $p^k(1-p)^{rk}$.

Therefore, we have to prove that the number of k-sets X that are independent sets in G_U is at least $1/e$ times the number of k-sets X that are independent sets in G_Z. We do this by picking X at random and showing that Prob[X is independent in G_U] $\geq e^{-1}$. We pick a random k-set vertex by vertex in G_U. Let v_i denote the ith vertex that is picked and V_i denote $\{v_1,\ldots,v_i\}$. The neighborhood set of V_i is denoted $N(V_i)$.

Thus, we have

$$\begin{aligned}
&\text{Prob}[V_k \text{ is independent}] \\
&= \prod_{j=1}^{k} \text{Prob}[v_j \notin N(V_{j-1}) \mid V_{j-1} \text{ is independent }] \\
&\geq \prod_{j=1}^{k}\left(1 - \frac{rk(j-1)}{8(s-j+1)}\right) \geq \prod_{j=1}^{k}\left(1 - \frac{2rk(j-1)}{8s}\right) \\
&\geq \left(1 - \frac{rk^2}{4s}\right)^k \geq e^{-1}.
\end{aligned}$$

The last inequality holds for sufficiently large n, since $k \ll d/rk^2$. Therefore, we conclude that $S_U \geq S_Z/e^2$.

We now complete the proof that Prob$[U \geq k] \geq$ Prob$[Z \geq k]/e^5$. Observe that

$$\text{Prob}[U \geq k] \geq S_U(1-p)^{s-k} \geq S_U/e^e \geq S_U/e^3.$$

The first inequality comes from the fact that the probability that there are exactly k isolated vertices in A is at least $S_U(1-p)^{s-k}$ (The lower bound is obtained by summing over all independent k-sets in G_U the probability that the k-set is isolated and all other vertices in A_U are not chosen). The second inequality uses the fact that $p < e/s$ and that for sufficiently large n, $k > e$, and thus $s-k < (s/e-1)e$. Therefore, to complete the proof of part 2,

$$\begin{aligned}\text{Prob}[U \geq k] &\geq \frac{1}{e^3}S_U \geq \frac{1}{e^5}S_Z\\ &\geq \frac{1}{e^5}\text{Prob}[Z \geq k].\end{aligned}$$

Proof of part 1. Let $\mathcal{I}$ be the set of independent sets in A of size at most $k/2$ whose weights sum to at least $k/2$. We want to relate the number of independent k-sets in A_U to the number of subsets in $\mathcal{I}$. To do that, we map every independent k-set X in A_U to an independent set in $\mathcal{I}$ as follows: if $|V_Y(X)| \geq k/2$, then map X to an arbitrary size-$k/2$ subset of $V_Y(X)$. Since the weight of a vertex is at least 1, the weight of the subset is at least $k/2$. If $|V_Y(X)| < k/2$, then map X to $V_Y(X)$. For any independent set $I \in \mathcal{I}$, there are at most $(k/8)^{k/2}\binom{s}{k/2}$ independent sets mapped to I: there are at most $(k/8)^{k/2}$ ways to pick the first $k/2$ vertices from the sets in G_U associated with the vertices of I and $\binom{s}{k/2}$ ways to pick the remaining vertices from A_U. Define

$$S_Y = \sum_{I \in \mathcal{I}} p^{|I|}(1-p)^{|N(I)|}.$$

We can lower bound S_Y by S_U/e as follows:

$$\begin{aligned}S_U &\leq \sum_{X \text{ indep., } |X|=k} p^k\\ &\leq \left(\frac{k}{8}\right)^{k/2}\binom{s}{k/2}\sum_{I\in\mathcal{I}}p^k\\ &\leq \left(\frac{k}{8}\right)^{k/2}\binom{s}{k/2}p^{k/2}\sum_{I\in\mathcal{I}}p^{k/2}\\ &\leq \left(\frac{k}{8}\right)^{k/2}\binom{s}{k/2}p^{k/2}\sum_{I\in\mathcal{I}}ep^{k/2}(1-p)^{kr/2}\\ &\leq \left(\frac{k}{8}\right)^{k/2}\binom{s}{k/2}p^{k/2}eS_Y.\end{aligned}$$

We show that

$$\left(\frac{k}{8}\right)^{k/2}\binom{s}{k/2}p^{k/2} \leq 1,$$

thus concluding that $S_Y \geq S_U/e$. Observe that p was chosen so that $p(1-p)^r = 4/d = 1/s$. So $p^{k/2} = (1/s)^{k/2}(1-p)^{-kr/2} \leq e(1/s)^{\frac{k}{2}}$ for sufficiently large n.

$$\left(\frac{k}{8}\right)^{k/2}\binom{s}{k/2}p^{k/2}$$

$$\leq \left(\frac{k}{8}\right)^{k/2} \frac{s^{k/2}}{(\frac{k}{2})!} es^{-k/2}$$
$$\leq \left(\frac{k}{8}\right)^{k/2} \left(\frac{2e}{k}\right)^{k/2} e \leq 1,$$

where the inequalities hold for sufficiently large n. In a similar manner to the proof of part 2,

$$\begin{aligned} \text{Prob}[Y \geq k/2] &\geq \frac{1}{e^3} S_Y \geq \frac{1}{e^4} S_U \\ &\geq \frac{1}{e^4}\text{Prob}[U \geq k]. \qquad \square \end{aligned}$$

This completes the proof of Lemma 9. We now proceed with the proof of Theorem 6.

Consider the following n by n bipartite graph H. There is an edge between a vertex x on the left and a vertex y on the right for every job that agent x sends to agent y when agent x is isolated. There is an edge from vertex x on the left to vertex x on the right for every job that x keeps when x is isolated. Note that the degree of a left node in H is exactly d. If a vertex is isolated, then it sends its jobs according to the edges in H. We examine the load due to isolated vertices only.

Our proof proceeds in three steps. First, we address two special cases: (i) there is a right vertex in H of degree at least βd; (ii) there are at least d left vertices in H that have an edge of multiplicity $\beta/32$ or more. Then, we prove the theorem for graphs H that do not fall into either of these categories.

LEMMA 10. *If H has a right vertex v of degree at least βd, then the expected number of jobs sent to v is at least* 4β.

Proof. Let $u_1, u_2, \ldots, u_k$ be the neighbors of v on the left. For $i = 1, 2, \ldots, k$, let W_i denote the multiplicity of the edge between u_i and v. $\sum W_i \geq \beta d$. Each of the u_i's is isolated with probability $p(1-p)^r$. Therefore, the expected number of jobs that v get is $p(1-p)^r \sum W_i \geq p(1-p)^r \beta d \geq 4\beta$. $\square$

LEMMA 11. *If there are at least d left vertices in H with an edge of multiplicity $\beta/32$ or more, then the expected maximum number of jobs an agent gets is at least $\beta/8e^3$.*

Proof. There is a set S of at least d vertices such that if any of them is isolated, some agent has a load of at least $\beta/32$. For $v \in S$, let E_v denote the event that v is isolated and the remaining vertices in S are not chosen. $\text{Prob}[E_v] \geq p(1-p)^{rk}(1-p)^{d/4e-1} \geq (4/de^3)$. (Recall that $p \leq 4e/d$ and for n sufficiently large $rk \ll 1/p$.) The probability that some vertex in S is isolated is at least $\sum_{v \in S} \text{Prob}[E_v] \geq 4/e^3$. Thus, the expected maximum load is at least $\beta/8e^3$. $\square$

We now consider the remaining case, where all right vertices in H have degree at most βd, and at most d left vertices in H have edges of multiplicity $\beta/32$ or more. Remove all the vertices on the left which have an edge of multiplicity at least $\beta/32$. There are at least $n - d$ left vertices remaining.

We need the following lemma.

LEMMA 12. *If the maximum degree of a right vertex in H is βd, then there exists a subgraph H' of H such that the following hold:*

1. *The number of right vertices in H' is at least d/r.*
2. *The degree of each right vertex in H' is at least $d/4$.*

3. *For every two distinct right vertices x and y in H', their neighborhood sets do not intersect.*

Proof. Let $c = 2\beta d^3/r(n-2d)$. Define a graph I whose vertices are a subset of the right vertices of H as follows. Remove all right vertices of degree less than $d/2$. Remove edges until each remaining right vertex has degree exactly $d/2$. Call the remaining graph H''. Connect two right vertices by an edge iff the size of the intersection of their neighborhood sets in H'' is at least c. The degree of any vertex in I is at most d^2/c. The number of right vertices in H that have degree at least $d/2$ is at least $(n-2d)/2\beta$ because the maximum degree is βd and the sum of degrees is $(n-d)d$. Therefore, there is an independent set of size at least $(n-2d)c/2\beta d^2 \geq d/r$ in I. Take a subset of size d/r of the independent set together with the neighborhood sets of these vertices in H'' and the connecting edges. What we get is a collection of d/r stars. Each right vertex is a root of one star and has degree $d/2$. Remove from this collection any left vertex that participates in more than one star. Since each right vertex removes from each other star at most c left vertices, the resulting stars are vertex disjoint and have minimum root degree of at least $d/2 - cd/r$ which is at least $d/4$ for sufficiently large n. □

The neighborhood sets of the right vertices of H' induce a collection of d/r disjoint sets of vertices in G. We call this collection of sets $\mathcal{A} = A_1, A_2, \ldots, A_{d/r}$. To each vertex v that belongs to such a set, we assign it a weight W_v which is equal to the multiplicity of its edge to its adjacent right vertex in H'. Since we have removed the vertices incident to edges of high multiplicity, the weight of any vertex is at most $\beta/32 \leq k/8$. Each set has a total weight of at least $d/4$. Let a_i be the sum of the weights of the isolated vertices in set A_i. We wish to show that the expected maximum over all a_i is in $\Omega(\beta)$.

For the remainder of the proof, "adjacency" refers to adjacency in the knowledge graph G. $N(X)$ denotes the neighborhood set of a set of vertices X. A subset X of vertices is isolated (not isolated/chosen/not chosen) if every vertex in the set is isolated (not isolated/chosen/not chosen). Let E_j denote the event $(a_1 < k/2) \wedge (a_2 < k/2) \wedge \cdots \wedge (a_j < k/2)$.

LEMMA 13. *Let* $1 \leq j \leq d/r$. *If* $\mathrm{Prob}[E_{j-1}] \geq 1 - (1/2e)$, *then*

$$\mathrm{Prob}[a_j \geq k/2 \mid E_{j-1}] \geq \frac{1}{2e^4}\mathrm{Prob}[a_j \geq k/2].$$

Proof. Let $A = A_1 \cup A_2 \cup \cdots \cup A_{j-1}$. Let $E = E_{j-1}$. For each vertex $v \in A_j$ we introduce W_v indicator variables as in the proof of Lemma 9. Let $Y_1, \ldots, Y_s$ be those variables. S_Y is defined as in Lemma 9. Using the arguments from the proof of Lemma 9, we have that $\mathrm{Prob}[a_j \geq k/2 \mid E] \geq E[S_Y \mid E]/e^3$. Also, $E[S_Y]$ is a trivial upper bound on $\mathrm{Prob}[a_j \geq k/2]$. Therefore, it is sufficient to prove that $E[S_Y \mid E] \geq E[S_Y]/2e$. Fix an arbitrary independent set $I \in \mathcal{I}$. We need to prove that $\mathrm{Prob}[I \text{ is isolated } \mid E] \geq \mathrm{Prob}[I \text{ is isolated}]/2e$.

The event that I is isolated happens if and only if I is chosen and $N(I)$ is not chosen. Therefore, $p^{|I|} \geq \mathrm{Prob}[I \text{ is isolated}]$. Also,

$$\begin{aligned}&\mathrm{Prob}[I \text{ is isolated } \mid E]\\ &= p^{|I|}\mathrm{Prob}[N(I) \text{ is not chosen } \mid E].\end{aligned}$$

Since

$$\begin{aligned}\mathrm{Prob}[N(I) \text{ is not chosen}] &= (1-p)^{|N(I)|}\\ &\geq (1-p)^{kr} \geq e^{-1},\end{aligned}$$

we have that

$$\begin{aligned}
&\text{Prob}[N(I) \text{ is not chosen } \mid E] \\
&\geq \text{Prob}[N(I) \text{ is not chosen } \wedge E] \\
&\geq \text{Prob}[N(I) \text{ is not chosen}] + \text{Prob}[E] - 1 \\
&\geq \frac{1}{2e}.
\end{aligned}$$

Putting these facts together, we get that

$$\text{Prob}[I \text{ is isolated } \mid E] \geq \frac{p^{|I|}}{2e} \geq \frac{\text{Prob}[I \text{ is isolated}]}{2e}. \qquad \square$$

We can now complete the proof of Theorem 6. The condition of Lemma 9 holds for sufficiently large n. If there exists j, $1 \leq j \leq d/r$, for which $\text{Prob}[E_j] < 1 - 1/2e$, then the expected maximum load is at least $k/4e \geq \beta/16e$. Otherwise, we may use Lemma 13 to get

$$\begin{aligned}
&\text{Prob}[\max a_j \geq k/2] \\
&= 1 - \text{Prob}[\forall j, a_j < k/2] \\
&= 1 - \prod_j \text{Prob}[a_j < k/2 \mid E_{j-1}] \\
&= 1 - \prod_j (1 - \text{Prob}[a_j \geq k/2 \mid E_{j-1}]) \\
&\geq 1 - \prod_j \left(1 - \frac{1}{2e^4}\text{Prob}[a_j \geq k/2]\right) \\
&\geq 1 - \prod_j \left(1 - \frac{1}{2e^{13}}\text{Prob}[Z \geq k/2]\right) \\
&\geq 1 - \left(1 - \frac{r}{2e^{13}d}\right)^{d/r} \geq 1 - e^{\frac{-1}{2e^{13}}},
\end{aligned}$$

where the last inequality holds for sufficiently large n. Therefore, $E[\max\{a_i\}] \geq (1-e^{-1/2e^{13}})(k/2) \geq (1-e^{-1/2e^{13}})(\beta/8) = \Omega(\beta)$. Recalling that the expected optimal cost is at most $8e$, the theorem follows. $\square$

3. Routing. We show $n(\log n + \log r)$-node networks with n input nodes where a maximum edge congestion of $\sqrt{n/r}$ can be guaranteed by a particular r-strategy. Theorem 16 proves this to be at the least nearly optimal.

THEOREM 14. *Divide all the input nodes into groups of r consecutive nodes. If each agent knows the destinations of the other input nodes in its group then we can route any $n \times n$ permutation on an $n(\log n + \log r)$-node network with maximum edge congestion $\sqrt{n/r}$.*

Proof. The network is derived from the Beněs network. We take the first $\log(n/r)/2$ levels of a $\log n$-dimensional Beněs network, then the middle $2 \log r$ levels of that network, then the next $\log(n/r)/2$ levels. As with global routing on a Beněs network, the bound on edge congestion is guaranteed even if there are two inputs for each source and two outputs for each destination. In the proof, we trace the selection of each path by describing the motion of a "packet" that moves along the path.

In every level, number each node from top to bottom $0, 1, \ldots, n-1$. The nodes in each level are divided into n/r groups of r nodes. The groups in the first level (the input level) determine the knowledge graph—each group is a clique in the knowledge graph.

For the first $\log(n/r)/2$ levels, each input picks the greedy path based on the first $\log(n/r)/2$ bits of the destination.

At this point, each packet reaches a place which differs with its input in only the first $\log(n/r)/2$ bits. There are n/r $\log r$-dimensional Beněs networks in the middle. The source nodes of each such subnetwork (i.e., the nodes at level $\log(n/r)/2+1$ of the whole network) receive packets from at most $\sqrt{n/r}$ different cliques. Furthermore, the place of each packet within its clique has not changed (that is, the place of each packet agrees with its source in the last $\log r$ bits). Thus, each node has at most two packets from each clique.

Now we will use the next $2\log r$ levels to route within each $r \times r$ network according to the last r bits of the destination (to be explained later).

Afterwards, the place of each packet corresponds with its destination in the first $\log(n/r)/2$ bits and the last $\log r$ bits. If the bits are numbered from left to right, then the place of a packet can only disagree with its destination in bits $\log n/r/2+1$ through $\log n/r$. Thus, there are at most $\sqrt{n/r}$ packets per node. Each packet then takes the greedy path to its destination. The congestion never exceeds $\sqrt{n/r}$ since at each successive level, the upper bound on the congestion decreases by a factor of two.

So far, everything specified about the paths can be determined by each source node without any extra information besides the destination of its own inputs.

We now have to explain how to do the routing in the middle $r \times r$ subnetworks. Now *source* and *destination* refer to the source and destination within the $r \times r$ network. We are guaranteed that for any such subnetwork, we only have $\sqrt{n/r}$ cliques routing simultaneously on that subnetwork. There are at most two packets per source from any single clique. There are at most $\sqrt{n/r}$ packets per destination from all cliques. Each clique determines the paths for all inputs in that clique so as to satisfy the conditions of Lemma 15 below. We then examine what happens when we have $\sqrt{n/r}$ cliques superimposed on the same $r \times r$ network.

In each level, number the nodes $0, 1, \ldots, r-1$ from top to bottom. Number levels from right to left so that the destination nodes are at level 1. For $j \in \{0, \ldots, 2^{\log r - k - 1} - 1\}$, define $S_i(j,k)$ to be the nodes in the ith level numbered $2^{\log r - k + 1}x + j$, for all $x \in \{0, 1, \ldots, 2^{k-1} - 1\}$. So, for any i, i' with $i > i'$ and any $k \geq i$, packets arriving at nodes in $S_i(j,k)$ can only reach level-i' nodes in $S_{i'}(j,k)$. In particular, packets arriving in $S_i(j,i)$ can only reach level-1 nodes in $S_1(j,i)$. We have the following lemma.

LEMMA 15. *Each clique can route the paths of its inputs so that the following hold:*

1. *In the first half of any particular $r \times r$ Beněs network, each edge carries at most one path.*

2. *In the second half of the network, the congestion along each edge coming into nodes in $S_i(j,i)$ from the previous level is the same (± 1).*

Proof. The proof is by induction on r. Consider the two $(r-1) \times (r-1)$ subnetworks in the middle. Suppose we manage to route the $r \times r$ problem so that each source sends one packet through the bottom network and one through the top network. And suppose that we manage to route the packets so that if a destination gets t packets, $t/2$ are routed through the bottom network and $t/2$ are routed through the top. (If t is not even, the number of packets routed through the top and the number routed through the bottom may differ by 1). Then we have managed to satisfy the constraints for the outer two level. By induction, we can satisfy the constraints for the inner levels.

Therefore, we just have to show that we can route the packets in that way. Make a bipartite graph—left vertices represent sources, right vertices represent destinations. An edge from a left vertex to a right vertex represents a path routed from that source to that destination. The degree of each vertex on the left is 2. The edges can be colored red and blue so that every vertex has half its edges red and half blue. The red edges represent paths through the upper subnetwork and the blue edges represent paths through the lower one. To see that the coloring of the edges can be done, split every vertex on the right with degree higher than 2 into vertices of degree 2 and at most one of degree 1. Now combine pairs of degree 1 vertices so that the graph is 2-regular. By Hall's theorem, the edges can be colored so that every vertex is incident to a red and a blue edge. When the vertices are recombined to get the original graph, if a vertex is incident to d edges, then at least $\lfloor d/2 \rfloor$ of the edges are colored with each color. □

We argue that if each clique can route its paths through each subnetwork so that conditions 1 and 2 are maintained, then the congestion is at most $\sqrt{n/r}$ even when we consider the congestion from all $\sqrt{n/r}$ cliques that are routing simultaneously on a particular subnetwork. In the first half of the network, each clique has only one path per edge. Thus, the total edge congestion is at most $\sqrt{n/r}$. In the second half of the network, paths that reach a node in $S_i(j,i)$ can only reach nodes in $S_1(j,i)$. If there are x paths coming into a node in $S_i(j,i)$, then property 2 guarantees that there are a total of $x2^{i-1}$ paths coming into all nodes in $S_i(j,i)$. These paths are all destined for nodes in $S_1(j,i)$. There can only be $2^{i-1}\sqrt{n/r}$ such paths because each destination gets only $\sqrt{n/r}$ paths from all cliques. Thus $x \leq \sqrt{n/r}$. □

THEOREM 16. *For every n-input, n-output, N-node, degree-d network, for every deterministic routing strategy with degree-r knowledge graph,*

1. *there exists a permutation for which the maximum congestion at a node is at least* $(1/2)\sqrt{n^2/Nr} - (n/2Nr)$;
2. *there exists a permutation for which the maximum congestion at an edge is at least* $(1/2d)\sqrt{n^2/Nr}$.

The proof is more or less a straightforward adaptation of the oblivious routing lower bounds of Borodin and Hopcroft [7] and Kaklamanis, Krizanc, and Tsantilas [14].

Proof: Part 1. Let S be an independent subset of size $n/2r$ in the knowledge graph. It has a neighbor set (in the knowledge graph) of size at most $n/2$. Fix the destinations of the sources in this neighbor set. We will ignore the congestion due to these paths. The algorithm for the inputs in S is now specified by a set of at least $n/2$ paths for each input—one to each remaining target.

Let $a = (1/2)\sqrt{n^2/(Nr(d+1))}$. We can assign to each of $t \geq (n/2) - da$ of the possible paths of an input node in S an internal node w such that each assigned w is assigned for at least a paths and at most $(d+1)a = (1/2)\sqrt{n^2/(Nr(d+1))}$ paths. The reason is as follows. Let $u \in S$. Repeat the following process. Find a node w that is an internal node of at least a paths from u. Mark it and assign w to a of these paths. Remove the paths and repeat till no such node can be found. Now, for each of the paths that have not been assigned a node, trace the path from its destination to its source, and assign it the first marked node that it encounters (excluding its source or destination) if there is one. A path does not get assigned a node either if its destination is a neighbor of u or if it passes through an unmarked neighbor of u. There are at most da such paths. The number of paths that get assigned any particular marked node w is at most the a paths that marked w plus any path that passed through an unmarked neighbor of w. There are at most da of the latter.

The number of distinct nodes w assigned for a fixed input in S is at least $(n/2 - da)/((d+1)a) \geq n/(2(d+1)a) - 1$. There are $n/2r$ nodes in S, so a total of $n^2/(4(d+1)ra) - (n/2r)$ nodes w are hit. Since there are N nodes in the network, there is a node w that is hit by at least $n^2/(4N(d+1)ra) - (n/2Nr) = (1/2)\sqrt{n^2/(Nr(d+1))} - (n/2Nr)$ paths from different input nodes. Each input node that hits w can choose from among at least $(1/2)\sqrt{n^2/Nr(d+1)}$ destinations that cause a hit on w, so there is a choice of distinct destinations for all input nodes that hit w.

Part 2. Let S be an independent set of size $t = n/2r$ in the knowledge graph. As in part 1, we fix the destinations of the sources in the neighborhood set of S and ignore the congestion due to the paths they choose. The destination of any source in S can be any of at least $n/2$ remaining output nodes in the network. An algorithm is specified by $nt/2$ paths. There are at least $t-1$ paths that end at any output node v (because v can also be a source node in S). Let $S(v)$ denote the set of edges in the network which have $k = \sqrt{n^2/Nr}/2d$ or more paths ending in v passing through them. Let $S^*(v)$ be the set of nodes incident to edges in $S(v)$. Note that $|S^*(v)| \leq 2|S(v)|$. Also, $v \in S^*(v)$. Thus,

$$|S - S^*(v)| \leq (k-1)d|S^*(v)|.$$

(For each source u not in $S^*(v)$, follow its path to v until it hits a node in $S^*(v)$. The last edge in this path has less than k paths in it.) This gives us that

$$t \leq kd|S^*(v)| \leq 2kd|S(v)|.$$

Thus $t/2kd \leq |S(v)|$. Summing over all $n/2$ destinations v,

$$\sum_{v \in V} |S(v)| \geq \frac{nt}{4kd}.$$

Since there are $Nd/2$ edges in the network, there is some edge e for which $e \in S(v)$ for at least

$$\frac{nt/4kd}{Nd/2} = k$$

different values of v.

Select e and $v_1, \ldots, v_k$ such that $e \in S(v_i)$ for $1 \leq i \leq k$. This means that we can find $u_1, \ldots, u_k$ such that $u_i \neq u_j$ for $i \neq j$ and the paths from u_i to v_i all pass through e. □

The following is a lower bound for randomized oblivious routing strategies. The lower bound holds when the knowledge graph is a set of fixed r-cliques.

THEOREM 17. *For every n-input, n-output, n-node, degree-d network, for every randomized routing strategy with a deterministic knowledge graph consisting of disjoint r-cliques, if $dr \leq n/2\log^4 n$, then there is a permutation for which the expected maximum congestion at a node is in $\Omega(\log(n/r)/\log\log(n/r))$.*

Proof. The proof follows a lower bound on randomized oblivious routing due to Borodin et al. [9]. We show a probability distribution over permutations that beats every deterministic routing strategy. Let $t = n/r$. There are t r-cliques in the knowledge graph. Each node will be numbered by a pair (i, j), where i represents the name of the clique the node is in and j represents the name of the node within the clique. $1 \leq i \leq t$ and $1 \leq j \leq r$. We pick one of $t!$ permutations uniformly at random. Each permutation is specified by a permutation over the cliques. If clique i is mapped to clique k, then (i, j) has to connect to (k, j) for all $1 \leq j \leq r$. Thus each node has to connect to one of t destinations. For some valid source–destination pairs (u, v), we assign a node (called $V(u, v)$) which is an internal node in the path from u to v. $V(C, D)$ denotes the multiset of nodes $V(u, v)$, where $u \in C$, $v \in D$, (u, v) is a valid source–destination pair, and $V(u, v)$ exists. $|V(C, D)|$ denotes the size of the set, counting multiplicity.

LEMMA 18. *Consider a clique C. The assignment can be picked so that the following hold:*

1. *A node w appears in at most $(d+1)\log t$ multisets $V(C, D)$.*
2. *If a node w appears in a multiset $V(C, D)$, then it appears in $V(C, D)$ for at least $\log t$ different cliques D.*
3. $\sum_D |V(C, D)| \geq n - dr\log t$.

Proof. Consider all the paths from a clique C to all the destination cliques D. There are a total of n paths. (For each source–destination pair of cliques, there are r paths and t destination cliques). Color each path so that the paths arriving at the same clique have the same color. This means that all the paths originating at a single source node have different colors. There are at most r paths colored with the same color. Now for each node that has at least $\log t$ paths of different colors going through it, mark the node and assign it $\log t$ paths of different colors. Then for each unassigned path, follow it from destination to source and assign it to the first marked node that it hits. A node will get at most $dr\log t$ paths of at most $d\log t$ colors this way. (This is because if a path gets assigned to a node, it came from an unmarked node. There are at most d unmarked nodes adjacent to a marked node. The paths that pass through an unmarked node have fewer than $\log t$ colors. There are at most r paths of a given color.)

How many paths don't get assigned? A path doesn't get assigned if it reaches its source node without hitting a marked node. There are r source nodes. How many make it back to a single source node u without getting assigned? If it doesn't get assigned, it passes through an unmarked neighbor of u just before it hits u. Call the neighbor v. All the paths going into u have different colors. If v is unmarked, than there are fewer than $\log t$ paths of different colors that pass through v. Thus there are fewer than $d\log t$ unassigned paths that reach u. □

In what follows, we denote by (C, D) the color of the r paths that lead from sources in C to destinations in D. Each of the t^2 combinations has a distinct color.

Now execute the following procedure. Find a node w that is hit by a single color (C, D) at least $\log t$ times. Remove all paths from C and repeat.

If at least $t/2$ such nodes are found, then we have the following situation: There is a sequence $C_1, C_2, \ldots, C_{t/2}$ of source cliques and a sequence $D_1, D_2, \ldots, D_{t/2}$ of destination cliques (not necessarily distinct) such that if D_i is assigned to C_i, there is a node that is $\log t$ congested. The probability that D_i is assigned to C_i is $1/t$. Conditioned upon D_1 not assigned to C_1, D_2 not assigned to $C_2, \ldots, D_{i-1}$ not assigned to C_{i-1}, the probability that D_i is assigned to C_i is still at least $1/2t$. Therefore, with probability at least $1-(1-1/2t)^{t/2} \geq 1-e^{-1/4}$, there is a $\log t$-congested node.

Otherwise, we have removed at most $t/2$ source cliques. The remaining source cliques have the property that no node is hit more than $\log t$ times by the same color. From now on, we consider only the remaining source cliques.

Let E_{Cw} denote the expected number of paths from sources in C that pass through w. The assignment of $V(C,D)$ has the property that $\sum_w \sum_C E_{Cw} \geq (n - dr\log t)/2$, where the second sum is taken over the remaining source cliques C.

Suppose that for all w, $\sum_C E_{Cw} \leq \log t$ (Otherwise, the lower bound on the expected congestion is established). Let $\delta = 1/8$. A node w is *good* if $\sum_C E_{Cw} \geq \delta$. The number of good nodes is at least $(n/2 - \delta n - dr \log t/2)/(\log t - \delta) \geq n/8 \log n$.

Fix a good node w. The assignment of $V(C,D)$ has the following property. For every clique C, either $E_{Cw} = 0$ or $\log t/t \leq E_{Cw} \leq ((d+1)\log^2 t)/t$. To see the upper bound, recall that none of the nodes are hit more than $\log t$ times by a single color $(C, \cdot)$. By claim 1 of Lemma 18, the upper bound follows. The lower bound follows from claim 2 of Lemma 18.

We want to lower bound the probability that h source cliques hit w for some $h < \log t$. Each hit corresponds to some clique C such that $E_{Cw} \geq \log t/t$. There are at least $\log t$ cliques D such that w is in $V(C,D)$. A hit by C removes one destination clique D from the list of possible destinations of other source cliques. Consider the conditional expectation of the number of paths from sources in C that pass through w, conditioned upon at least h other source cliques having hit w. By the above arguments, this conditional expectation is at least $1/t$. Notice further that if $w \in V(C,D)$, it may appear in $V(C,D)$ at most $\log t$ times (because of our assumption on the remaining source cliques). If D is assigned to C, we will count this as only one hit, so the expected number of hits on w is bounded below by $1/\log t \sum_C E_{Cw}$. Also, the probability of a hit due to source C is at most $(d+1)\log t/t$.

We use the following probabilistic lemma, due to Borodin et al. [9].

LEMMA 19 (Borodin et al.). *Let* $x_1, \ldots, x_k$ *be independent* $0/1$ *random variables. Let* $p_i = \text{Prob}[x_i = 1]$, $\sum p_i = \sigma$, *and* $a \leq p_i \leq b$. *Then, for any* $Y \leq \sigma/(2eb\log(b/a))$,

$$\text{Prob}\left[\sum x_i > Y\right] \geq \left(\frac{\sigma}{2e\log(b/a)Y}\right)^Y.$$

In our case, $\sigma = \delta/\log t = 1/8\log t$, $a = 1/t$, *and* $b = (d+1)\log t/t$.

According to the lemma, for n sufficiently large, the probability that node w is at least $Y = \log t/8 \log\log t$ congested is at least $1/\sqrt{t} \geq \log t/t$, provided that the condition on Y holds. It can be easily verified that for sufficiently large n, the condition that $dr \leq n/2\log^4 n$ implies the required condition on Y.

To summarize, we have shown that one of the following properties holds:

1. there is a sequence of (not necessarily distinct) nodes $x_1, x_2, \ldots, x_{t/2}$ such that with constant probability, at least one of them is at least $\log t$ congested; or
2. there is a node w whose expected congestion is at least $\log t$; or
3. there is a node w_1 that is at least $\log t/8\log\log t$ congested with probability at least $1/\sqrt{t} \geq \log t/t$.

If the first or second case holds, we are done. If the third case holds, we would like to construct a sequence of $t/\log t$ nodes $w_1, w_2, \ldots, w_{t/\log t}$ with the property that for every i, if $w_1, w_2, \dots, w_{i-1}$ are less than $\log t/\alpha \log\log t$ congested, then w_i is at least $\log t/\alpha \log\log t$ congested with probability at least $\log t/t$ for some constant α. We construct such a sequence by repeating the entire argument above at most $t/\log t$ times. If $w_1, w_2, \dots, w_i$, $i < t/\log t$, are less than $\log t/\log\log t$ congested, this constrains $t' = i \log t/\log\log t < t/\log\log t$ source cliques to their destination cliques. We remove these cliques and can thus repeat the above discussion setting $t := t'' = t - t'$ and assigning new values $V(u,v)$ for the remaining cliques. Notice that $t'' \geq t(1 - 1/\log\log t)$. We get one of the following properties:

1. there is a sequence of (not necessarily distinct) nodes $y_1, y_2, \dots, y_{t''/2}$ such that with constant probability, at least one of them is at least $\log t'' \geq \log t - 1$ congested; or
2. there is a node w' whose expected congestion is at least $\log t'' \geq \log t - 1$; or
3. there is a node w_{i+1} that is at least $\log t''/8 \log\log t'' \geq (\log t - 1)/8 \log\log t$ congested with probability at least $1/\sqrt{t''} \geq \log t/t$.

All the probabilities and expectations are conditioned upon the assignment of destination cliques to the t' constrained source cliques.

In the first case, recall that it followed from each of the y's being hit by $\log t''$ paths of one pair of source–destination cliques. If we remove the conditioning upon the assignment of destination cliques to the t'-constrained source cliques, this at most halves the probability of such a hit (because $t' < t/2$), so there would still be a constant probability for one of the y's to be at least $\log t - 1$ congested. In the second case, a similar argument shows that removing the condition at most halves the expected congestion at w' (using linearity of expectations). In the third case, we have constructed another node in the sequence $w_1, w_2, \dots, w_{t/\log t}$ and may proceed to constructing the next one. □

In the above proof, the permutation depends on the r-cliques. Assuming that $dr^3 \leq n/\log^3 n$, the lower bound in fact holds even when the agents can organize themselves into random r-cliques. The proof requires choosing uniformly at random an n-permutation of destinations rather than mapping cliques to cliques. We omit the proof.

4. Open problems.

We have given tight bounds, up to constant factors, for c_r, both in the deterministic and the randomized cases. It would be interesting to give better bounds for c_G in terms of other parameters of the graph G. Given G, what is the computational complexity of determining c_G? Our randomized upper bounds use shared coins. Can the same bounds be achieved with private coins? Our bounds for routing are far from being tight. Can one route deterministically on a high-degree network (where d is not a constant) achieving node or edge congestion close to the lower bounds given in Theorem 16? Does the randomized lower bound for node congestion hold for arbitrary knowledge graphs? Can a matching upper bound be achieved for large r?

Acknowledgments. We would like to thank the referees for their thorough reading and useful comments and suggestions. In particular, we would like to thank referee B for pointing out a flaw in our original proof of Lemma 9.

REFERENCES

[1] A. Aggarwal, A. K. Chandra, and M. Snir, *Communication complexity of PRAM's*, Theoret. Comput. Sci., 71 (1990), pp. 3–28.

[2] N. Alon, private communication.

[3] B. Awerbuch, O. Goldreich, D. Peleg, and R. Vainish, *A tradeoff between information and communication in broadcast protocols*, J. Assoc. Comput. Mach., 37 (1990), pp. 238–256.

[4] V. Beněs, *Mathematical Theory of Connecting Networks and Telephone Traffic*, Academic Press, New York, 1965.

[5] B. Bollobás, *Random Graphs*, Academic Press, New York, 1985.

[6] A. H. Bond and L. Gasser, eds., *Readings in Distributed Artificial Intelligence*, Morgan Kaufmann, San Francisco, 1988.

[7] A. Borodin and J. Hopcroft, *Routing, merging, and sorting on parallel models of computation,* J. Comput. System Sci., 30 (1985), pp. 130–145.

[8] A. Borodin, N. Linial, and M. Saks, *An optimal on-line algorithm for metrical task systems.* J. Assoc. Comput. Mach., 39 (1992), pp. 745–763.

[9] A. Borodin, P. Raghavan, B. Schieber, and E. Upfal, *How much can hardware help routing?*, in Proc. 25th ACM Symposium on the Theory of Computing, San Diego, Association for Computing Machinery, New York, 1993, pp. 573–582.

[10] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, T. von Eicken, *LogP: Towards a realistic model of parallel computation,* in SIGPLAN Notices, vol. 28, Special Interest Group on Programming Languages, Association for Computing Machinery, New York, 1993, pp. 1–12.

[11] X. Deng and C. H. Papadimitriou, *Competitive distributed decision making,* International Federation for Information Processing, A-12 (1992), pp. 250–256.

[12] C. Dwork, M. Herlihy, and O. Waarts, *Contention in shared memory algorithms,* in Proc. 25th ACM Symposium on Theory of Computing, San Diego, Association for Computing Machinery, New York, 1993, pp. 174–183.

[13] R. L. Graham, *Bounds on multiprocessing timing anomalies,* SIAM J. Appl. Math., 17 (1969), pp. 416–429.

[14] C. Kaklamanis, D. Krizanc, and T. Tsantilas, *Tight bounds for oblivious routing on the hypercube,* Math. Systems Theory, 24 (1991), pp. 223–232.

[15] A. R. Karlin, M. S. Manasse, L. Rudolph, and D. D. Sleator, *Competitive snoopy caching,* Algorithmica, 3 (1988), pp. 70–119.

[16] C. H. Papadimitriou and M. Yannakakis, *Towards an architecture-independent analysis of parallel algorithms,* SIAM J. Comput., 19 (1990), pp. 322–328.

[17] ——— *Linear programming without the matrix,* in Proc. 25th ACM Symposium on Theory of Computing, San Diego, Association for Computing Machinery, New York, 1993, pp. 121–129.

[18] D. D. Sleator and R. E. Tarjan, *Amortized efficiency of list update and paging rules,* Comm. Assoc. Comput. Mach., 28 (1985), pp. 202–208.

[19] L. G. Valiant, *A bridging model for parallel computation,* Comm. Assoc. Comput. Mach., 33 (1990), pp. 103–111.

[20] A. Waksman, *A permutation network,* J. Assoc. Comput. Mach., 15 (1968), pp. 159–163.

[21] A. C. Yao, *Probabilistic computations: Toward a unified measure of complexity,* in Proc. 18th IEEE Symposium on Foundations of Computer Science, IEEE Press, Piscataway, NJ, 1977, pp. 222–227.

SIAM J. COMPUT.
Vol. 25, No. 3, pp. 520–539, June 1996

COMPLEXITY OF SUB-BUS MESH COMPUTATIONS*

ANNE CONDON†, RICHARD LADNER‡, JORDAN LAMPE§, AND RAKESH SINHA¶

Abstract. The time complexity of several fundamental problems on the sub-bus mesh parallel computer with p processors is investigated. The problems include computing the PARITY and MAJORITY of p bits, the SUM of p numbers of length $O(\log p)$, and the MINIMUM of p numbers. It is shown that in one dimension the time to compute any of these problems is $\Theta(\log p)$. In two dimensions the time to compute any of PARITY, MAJORITY, and SUM is $\Theta(\frac{\log p}{\log\log p})$. It was previously shown that the time to compute MINIMUM in two dimensions is $\Theta(\log\log p)$ [R. Miller et al., *IEEE Trans. Comput.*, 42 (1993), pp. 678–692; L. Valiant, *SIAM J. Comput.*, 4 (1975), pp. 348–355]

Key words. sub-bus mesh, reconfigurable mesh, time complexity, parity, majority, sum, minimum, CRCW, PRAM, complexity

AMS subject classifications. 68Q10, 68Q15, 68Q22, 68Q25

1. Introduction.

A sub-bus mesh computer is a single-instruction multiple-data (SIMD) two-dimensional array of processors, where processors can broadcast data vertically or horizontally on segmented busses. On a segmented bus, some of the processors on the bus are active while others are inactive. Each active processor can broadcast on the bus to all processors up to the next active processor. Thus, all the intervening inactive processors receive the data that has been broadcast. The sub-bus mesh computer architecture has been implemented on the commercially available MasPar MP-1 [5].

Typically, there are p processors in a $\sqrt{p}\times\sqrt{p}$ two-dimensional array, where $\sqrt{p}$ is a power of two. We will also consider one-dimensional meshes. For each of the problems we will consider, we assume there are p inputs, distributed one per processor.

The purpose of this paper is to present upper and lower bounds on the time to compute several fundamental functions including the PARITY and MAJORITY of p bits, the SUM of p numbers of length $O(\log p)$, and the MINIMUM of p numbers. Reisis and Prasanna Kumar appear to be the first to consider efficient algorithms for the sub-bus mesh architecture [26]. To our knowledge, this is the first paper to prove extensive results on the power and limitations of the sub-bus computer architecture.

1.1. Results.

There are simple algorithms for computing the Boolean OR and AND in constant time on a one- or two-dimensional sub-bus mesh computer [26].

* Receved by the editors October 12, 1993; accepted for publication (in revised form) September 20, 1994.

† Computer Sciences Department, University of Wisconsin, Madison, WI 53706. The research of this author was supported by National Science Foundation grants CCR-9100886 and CCR-9257241 and by matching funds from Digital Equipment Corporation and the AT&T Foundation.

‡ Department of Computer Science and Engineering, University of Washington, Seattle, WA 98195. The research of this author work was supported by National Science Foundation grant CCR-9108314. Part of this author's research was done while he was visiting Victoria University of Wellington, New Zealand.

§ Department of Computer Science and Engineering, University of Washington, Seattle, WA 98195. The research of this author work was supported by National Science Foundation grant CCR-9108314.

¶ Department of Computer Science and Engineering, University of Washington, Seattle, WA 98195. Current address: School of Computer Science, Florida International University, University Park, Miami, FL 33199. The research of this author was supported by National Science Foundation/Defense Advanced Research Projects Agency grant CCR-8907960 and National Science Foundation grant CCR-8858799.

Two other natural Boolean functions are PARITY and MAJORITY. We show that PARITY and MAJORITY are computable in time $\Theta(\log p)$ on a one-dimensional sub-bus mesh. The proofs of the lower bounds use a new technique for bounding the amount of information about the input that can be distributed on a one-dimensional mesh. The proof of the lower bound for MAJORITY can be used to show lower bounds for some other symmetric Boolean functions. We also consider the problem of computing the MINIMUM on a one-dimensional sub-bus mesh, and show that MINIMUM is computable in $\Theta(\log p)$ time. The lower bound for MINIMUM on the one-dimensional sub-bus mesh uses the same technique as the lower bound for PARITY.

We then show that PARITY and MAJORITY are computable in time $\Theta(\frac{\log p}{\log\log p})$ on a two-dimensional sub-bus mesh computer. The lower bounds follow from the fact that a concurrent-read/concurrent-write parallel random-access machine (CRCW PRAM) can simulate a sub-bus mesh computer to within a constant factor of the time and within a polynomial number of processors. Thus, the CRCW PRAM lower bounds on PARITY and MAJORITY [4] apply to the sub-bus mesh computer. The upper bounds follow from an algorithm for SUM, the sum of p numbers of length $O(\log p)$, which runs in time $O(\frac{\log p}{\log\log p})$. The SUM algorithm is nontrivial, using mixed radix arithmetic, the Chinese remainder theorem, and recursion to achieve the result. The obvious algorithm for SUM takes time $\Theta(\log p)$. The two-dimensional bound of $\Omega(\frac{\log p}{\log\log p})$ for SUM on the CRCW PRAM follows from the lower bound on PARITY.

1.2. Related results. The mesh or array parallel computer architecture has been investigated for a number of years, with numerous articles on its many variants [13, 17, 18, 22, 21, 26, 29]. The sub-bus mesh architecture was first investigated by Reisis and Prasanna Kumar [26], where they gave constant time algorithms for the OR of p bits and the MINIMUM of $\sqrt{p}$ numbers (all on one row), and an $O(\log p)$ algorithm for combining p data items with an associative operator. Two variants of the mesh computer are closely related to the sub-bus mesh. First, there is the full-bus mesh where processors can broadcast vertically or horizontally, but on a vertical (horizontal) broadcast at most one processor per column (row) can be active. The MPP of Goodyear and NASA is an example of a full-bus two-dimensional mesh computer [3]. Full-bus meshes are generally less powerful than sub-bus meshes. Both PARITY and MINIMUM require $\Omega(p^{\alpha})$ time for some $\alpha > 0$ on full-bus meshes [2, 25].

Second, there is the reconfigurable mesh, which allows the topology of the mesh to be changed by the executing program [18]. Several prototype, but no commercial, reconfigurable mesh computers have been built. PARITY can be computed in constant time on the reconfigurable, two-dimensional mesh [18]. Thus, our results demonstrate that the sub-bus mesh computer architecture is strictly more powerful than the full-bus mesh computer architecture, but strictly less powerful than the reconfigurable mesh computer. In other work on the PARITY function, MacKenzie [19] independently obtained a lower bound of $\Omega(\log p/k)$ for computing PARITY on a restricted $p \times k$ reconfigurable mesh model, which is exactly our one-dimensional sub-bus mesh model for $k = 1$. However, he did not extend this to other symmetric functions.

The SUM function has also been previously studied on the reconfigurable mesh. Nakano [23] and Nakano, Masuzawa, and Tokura [24] developed algorithms for SUM on the reconfigurable mesh that also use Chinese remaindering. Their results do not apply directly to the sub-bus mesh architecture. MINIMUM has also been previously studied on the two-dimensional reconfigurable mesh, and the techniques used can be applied directly to show that MINIMUM can be computed in $\Theta(\log\log p)$ time on the two-dimensional sub-bus mesh. From the work of Hao, MacKenzie, and Stout [12], a lower bound of $\Omega(\log\log p)$ is obtained for computing MINIMUM on a two-dimensional sub-bus mesh. Their proof is based on a PRAM simulation of the mesh model, and applies a result of Fich et al. [11] in which an equivalent lower bound is proved for the CRCW PRAM. However, this proof requires that the inputs be very large. Another lower bound of $\Omega(\log\log p)$ for computing MINIMUM on the two-dimensional sub-bus mesh follows from the general lower bound for the parallel comparison model of Valiant [31] and applies to comparison-based algorithms. A matching upper bound is due to Miller et al. [21], and is basically an implementation of the parallel MINIMUM algorithm of Valiant [31].

The PRAM is probably the most well-studied theoretical model of parallel computation. There are a number of variants of the PRAM, depending on whether reads or writes to the same memory location can be done concurrently. The variant most closely related to the sub-bus mesh is the CRCW PRAM (see [14] for an introduction to the PRAM model). In this version more than one processor can read or write the same memory location at the same time. A simultaneous write can be resolved in a number of ways. The ability of the sub-bus mesh to broadcast on segments of the bus is very much like a combination of a concurrent read and a concurrent write. Those processors that are actively broadcasting are executing a concurrent write while those that are inactive are executing a concurrent read. Indeed the sub-bus mesh can compute OR in constant time just as a CRCW PRAM can. We show that CRCW PRAM can simulate a sub-bus mesh computer to within a constant factor of the time. This simulation immediately implies that lower bounds for the CRCW PRAM are also lower bounds for a sub-bus mesh. Interestingly, some CRCW PRAM algorithms can be translated into sub-bus mesh algorithms. For example, the constant-time OR algorithm and Valiant's MINIMUM algorithm can be implemented on the sub-bus mesh. By contrast, some CRCW PRAM algorithms, such as the constant-time CRCW PRAM MINIMUM algorithm of Fich, Radge, and Wigderson [10], appear to be impossible to implement on the sub-bus mesh. More generally, the CRCW PRAM is strictly more powerful than the sub-bus mesh regardless of the number of dimensions. This is because problems like SORT require time $\Omega(p^\alpha)$ on mesh computers, but can be done in time $O(\log p)$ on a PRAM.

The sub-bus model is also incomparable with the exclusive read, exclusive write (EREW) PRAM model. To see this, note that computing the OR of p bits requires $\Omega(\log p)$ time on a CREW PRAM [9], whereas the sub-bus mesh can compute OR in constant time. By contrast, p integers can be sorted in $O(\log p)$ time on an EREW PRAM with p processors [1, 7, 16], but, by a simple bisection bandwidth argument, this task requires time $\Omega(p^\alpha)$ on mesh computers [30].

1.3. Organization of the paper.

In §2 we present our model of the sub-bus mesh computer. In §3 we prove our upper and lower bounds for the one-dimensional sub-bus mesh computer. In §4 we give two-dimensional algorithms for PARITY and SUM. In §5 we give our simulation of a two-dimensional sub-bus mesh by a CRCW PRAM, thereby yielding our two-dimensional lower bounds for PARITY, MAJORITY, and SUM. In §6 we present two-dimensional upper and lower bounds for MINIMUM. Finally, we have our conclusions in §7.

2. Sub-bus mesh computer model.

For the purposes of this paper we present a simple version of the sub-bus mesh computer architecture. Actual machines have a richer organization.

The sub-bus architecture can be easily explained for the one-dimensional mesh or linear array of processors. There are p processors, numbered consecutively 0 to $p-1$ on a circle. Processor 0 is the *front-end* which runs the parallel program. Each processor is a random-access machine (RAM) with its own memory which is referenced using *plural* variables. In addition, there are *singular* variables for which there is only one copy which is stored at the front-end processor. There is a special plural variable `PID` that always holds the processor's number. Processor operations include direct and indirect Boolean operations, arithmetic operations, shifts, and comparisons. In addition, the front-end can perform normal branching operations and issue *parallel instructions.*

A parallel instruction issued by the front-end has the form "**if** <*condition*> **then** <*statement*>." Each processor evaluates the condition, which can be any sequence of nonbranching operations on plural or singular data that evaluates to a Boolean value. If the condition is true then the processor is said to be *active*; otherwise it is said to be *inactive.* Only the active processors execute the statement part of the instruction.

There are two kinds of statements, *local operations* and *segmented broadcasts.* A local operation is just a typical nonbranching RAM operation executed on plural or singular data at each processor. A segmented broadcast has the form

`broadcast_`*direction*`[`*distance*`].`*plural variable* ← *plural variable.*

The *direction* can be either `left` or `right`. The variable *distance* must be singular. When an active processor i executes the instruction `broadcast_right[`d`].y` ← `x` then the location `y` at the processors $(i+1) \bmod p, (i+2) \bmod p, \ldots, (i+j) \bmod p$ receive the value stored in location `x` of processor i, where processors $(i+1) \bmod p, (i+2) \bmod p, \ldots, (i+j-1) \bmod p$ are inactive and either $j = d$ or $j < d$ and processor $(i+j) \bmod p$ is active. The segmented broadcast to the left is similar. In either case, the circle is partitioned into nonoverlapping segments. Each segment behaves like a sub-bus of the bus that includes all the processors. The MasPar MP-1 implements the segmented broadcast as `xnetc`. Table 1 describes the result of a segmented broadcast.

TABLE 1
*Demonstration of segmented broadcast. The * indicates that the value of* `y` *did not change because of the broadcast.*

	`broadcast_right[2].y ← x`							
PID	0	1	2	3	4	5	6	7
active	no	no	yes	yes	no	no	no	yes
x	a	b	c	d	e	f	g	h
y	h	h	*	c	d	d	*	*

In two dimensions there are also p processors, where p is a square number. The mesh processors are arranged in a $\sqrt{p} \times \sqrt{p}$ array. The coordinates of a mesh processor's number are stored in `PIDx` and `PIDy`. A mesh processor's number is stored in `PID` = `PIDy` * $\sqrt{p}$ + `PIDx`. The sub-busses go in four directions: `up`, `down`, `right`, and `left`. Processor (x, y) is immediately up from processor $(x, (y+1) \bmod \sqrt{p})$ and immediately to the left of processor $((x+1) \bmod \sqrt{p}, y)$. So vertical busses go up and down, while horizontal busses go right and left, all in a circular fashion. The front-end processor is processor $(0, 0)$.

2.1. Time of sub-bus mesh algorithms. For the purpose of analyzing our algorithms we consider time to be evaluated using the unit-cost RAM criterion where the values operated upon must have length $O(\log p)$. Each sequential operation by the front-end, each parallel operation used in evaluating the condition in a parallel instruction, and each statement of a parallel instruction costs 1 in our model. We do not charge for the broadcast of parallel instructions by the front-end to the mesh processors. We assume that cost is dominated by the cost of executing the parallel instruction.

As mentioned earlier, the RAM instructions include the usual direct and indirect Boolean operations, arithmetic operations, shifts, and comparison. In addition, we permit any fixed finite set of RAM instructions for our processors, provided each instruction can be implemented in uniform NC [8, 28], that is, each instruction can be built from $\log^{O(1)} p$ hardware and runs in time $\log^{O(1)} \log p$. The set of RAM operations is independent of p. Thus, the total hardware in the sub-bus mesh computer of p processors is $p \log^{O(1)} p$. The running time of $\log^{O(1)} \log p$ per RAM instruction is fast enough to be considered constant time in the mesh of processors.

For the purpose of proving our lower bounds we allow our model to be even more general. We do not restrict the length of the values operated on and do not restrict the RAM operations in any way. There is one exception. The two-dimensional lower bound for MINIMUM is done in the so-called "comparison model," where the only RAM operations allowed on input values are comparison, copy, and broadcast. Thus, with one exception, the lower bounds reflect the cost of computing functions due to the sub-bus mesh architecture, not any limitations on the individual processors. The two-dimensional lower bound for MINIMUM is still quite general, but is limited to the comparison model of the sub-bus mesh computer.

2.2. Examples of sub-bus mesh algorithms. Below we give two examples of sub-bus mesh algorithms, both of which will be building blocks in subsequent algorithms. Both examples can be found in the paper by Reisis and Prasanna Kumar [26]. In our terminology, if x is a plural variable then we indicate its value at processor i by $\mathtt{x}_i$ or at processor (i, j) by $\mathtt{x}_{i,j}$.

Our first program computes the OR in constant time.

```
CONSTANT-TIME-OR on a one-dimensional mesh
input: plural Boolean x
output: OR of all values $x_i$ in plural variable y
begin
  y ← false
  if x = true then
    broadcast_left[$p$].y ← true
end
```

In CONSTANT-TIME-OR, if any of the values $\mathtt{x}_i$ are true, then one or more of the processors will make sure to broadcast true into all processors' y's. However, if *none* of the $\mathtt{x}_i$ bits are true, then no processor will run the broadcast step, and so all the $\mathtt{y}_i$'s will remain false. Clearly, using DeMorgan's law, AND can also be computed in constant time.

Our second program computes the MINIMUM of $\sqrt{p}$ values in constant time on a two-dimensional $\sqrt{p} \times \sqrt{p}$ mesh.

```
CONSTANT-TIME-MINIMUM of $\sqrt{p}$ values on a two-dimensional mesh
input: plural integer/real variables $x_{0,0}, \ldots, x_{\sqrt{p}-1,0}$
output: MINIMUM value $x_{i,0}$ in plural variable y
other: plural Boolean t
```

```
begin
  if PIDy = 0 then
    broadcast_down[$\sqrt{p}-1$].x ← x
  if PIDx = PIDy then
    broadcast_left[$\sqrt{p}$].y ← x
  t ← x > y
  if t then
    broadcast_up[$\sqrt{p}-1$].t ← true
  if not t then
    broadcast_left[$\sqrt{p}$].y ← x
end
```

In CONSTANT-TIME-MINIMUM, the first two broadcasts have the effect of setting $x_{i,j} = x_{i,0}$ and $y_{i,j} = x_{j,0}$. The comparison $x_{i,j} > y_{i,j}$ is then equivalent to the comparison $x_{i,0} > x_{j,0}$. If such a comparison holds then $x_{i,0}$ is not the minimum. The statement "`if t then`..." computes, in one step, the "or" of the outcomes of these comparisons. Thus, after the broadcast up, if $t_{i,0} =$ `false` then $x_{i,0}$ is the minimum. This minimum is then broadcast to the first row of the mesh.

3. One-dimensional bounds.

In this section we give precise upper and lower bounds on the parallel time to compute PARITY, MAJORITY, SUM, and MINIMUM on the one-dimensional sub-bus mesh computer.

3.1. Upper bounds in one dimension.

As observed by Reisis and Prasanna Kumar [26], all our problems can be computed by a one-dimensional algorithm which works for any associative binary operator and runs in $O(\log p)$ time. Let $\oplus$ be any binary associative operation. The value REDUCE-$\oplus$(x) is $x_0 \oplus x_1 \oplus \cdots \oplus x_{p-1}$ stored in processor 0. The following algorithm simply computes the expression for REDUCE-$\oplus$(x) as a balanced binary tree.

```
REDUCE-⊕ on a one-dimensional mesh
input: plural variable x.
output: $y_0 = x_0 \oplus x_1 \oplus \ldots \oplus x_{p-1}$
other: plural variable z, singular integer i
begin
  y ← x
  i ← 1
  while i < $p$ do begin
    if PID mod i = 0 then
      broadcast_left[i].z ← y
    if PID mod 2i = 0 and PID + i < $p$ then
      y ← y ⊕ z
    i ← i * 2
  endwhile
end
```

Both SUM and MINIMUM can be expressed as REDUCE-$\oplus$ operations, where $\oplus$ is integer addition in the case of SUM and the minimum of two numbers in the case of MINIMUM. PARITY and MAJORITY are easily computable in constant time from SUM. Thus we have the following theorem.

THEOREM 3.1 (see [26]). *On a one-dimensional sub-bus mesh with p processors,* PARITY, MAJORITY, SUM, *and* MINIMUM *can be computed in time $O(\log p)$.*

3.2. Lower bounds in one dimension. Our lower bounds for the one-dimensional sub-bus mesh computer are based on the limited communication bandwidth of this architecture. Thus, in proving our lower bounds, we use a simplified model in which internal computations in a processor are "free" and only the time taken for communication is measured. It will be clear that any lower bound for this model applies also to the upper-bound model.

As before, there are p processors, numbered $0, 1, \dots, p-1$, connected by a circular sub-bus. The computation proceeds in rounds; we charge 1 time unit for a round. In each round of the computation, the processors first communicate and then perform arbitrary internal computation. The communication is controlled by the front-end, processor 0, just as in the upper-bound model described in §2. Once this is done, processors can do arbitrary internal computation that does not require communication. There is no bound on the length of values broadcast or computed by the processors.

An algorithm consists of both the algorithm that determines the sequence of communication instructions broadcast by processor 0, and the algorithms of processors $0, \dots, p-1$ that determine the internal computations at each round. Since processor 0 can read any information on the bus that passes in either direction between processor $p-1$ and processor 1, the communication instructions may depend on this information.

Let f be a function with domain D^p. We say algorithm A computes f if for all $(x_0, x_1, \dots, x_{p-1}) \in D^p$, if at the start of the algorithm each processor i has in its memory the value x_i, then at the end of the algorithm, every processor has in a special memory location the value $f(x_0, x_1, \dots x_{p-1})$. The tuple $(x_0, x_1, \dots, x_{p-1})$ is called the input. In all of the results of this section, we assume that $|D| \geq 2$.

Fix an input $\boldsymbol{x} = (x_0, x_1, \dots, x_{p-1})$. Processor k's *view* on input $\boldsymbol{x}$ at time t is a sequence $k, x_k, (1, v_1), (2, v_2), \dots, (t, v_t)$, where v_i is the value received by processor k at time i during the broadcast instruction. String v_i is a special symbol, say ϵ, if no value is received. We denote by $\text{View}_k(\boldsymbol{x}, t)$ the view of processor k at time t. For a fixed input $\boldsymbol{x}$, we say x_i is *unknown to* processor k at time t if $\text{View}_k(\boldsymbol{x}, t) = \text{View}_k(\boldsymbol{x}', t)$ for all $\boldsymbol{x}'$ that differs from $\boldsymbol{x}$ only at component i. Otherwise, we say x_i is known to processor k at time t.

Our main result is the following theorem.

THEOREM 3.2. *On a one-dimensional sub-bus mesh with p processors and for any algorithm A, there exists an input* $\boldsymbol{x}$ *such that for some* $i, 1 \leq i \leq p-1$, x_i *is unknown to processor* 0 *at time* $\log p - 1$.

This result is true, regardless of what function is being computed by A, as long as the domain size $|D| \geq 2$. Hence, the result immediately yields a lower bound of $\log p$ for the time to compute functions f with the property that for any $(x_0, x_1, \dots, x_{p-1}) \in D^p$ and any $i, 0 \leq i \leq p-1$, there is some $x'_i \in D$ such that

$$f(x_0, x_1, \dots, x_{i-1}, x_i, x_{i+1}, \dots, x_{p-1}) \neq f(x_0, x_1, \dots, x_{i-1}, x'_i, x_{i+1}, \dots, x_{p-1}).$$

Clearly PARITY is an example of such a function, where $D = \{0, 1\}$, and so Theorem 3.2 implies a lower bound of $\log p$ for PARITY. Also, the MINIMUM function over the integers is an example of such a function, so again Theorem 3.2 implies a lower bound of $\log p$ for computing MINIMUM.

We now describe informally the ideas in the proof of Theorem 3.2. Note that, for all inputs $\boldsymbol{x}$ and processors k, if $i \neq k$ then x_i is unknown to k at time 0. Consider a processor i at the first round. We consider two possibilities. The first is that i is inactive at round 1, regardless of its input value x_i. This is good since then, for any processor $k \neq i$, x_i is still unknown to processor k at time 1.

The other possibility is that i is "potentially active," that is, i is active on at least one possible value of its input. Then, unfortunately, at the end of round 1, x_i may be known to some, and possibly all, other processors. We can use this to our advantage, however, by setting i's input x_i to force i to be active. Then, the broadcast of processor i will block any other broadcast that might otherwise have sent information through i.

For the purpose of this informal discussion, suppose that at round 1 all processors are potentially active. Our strategy in this case will be to fix the values of alternate processors, in order to force them to be active. These fixed values determine a partial assignment $\alpha \in (D \cup \{*\})^p$, and partition the processors into *fixed* and *free* processors. On any input $\boldsymbol{x}$ consistent with the partial assignment α, the broadcasts of the fixed, active processors block the free processors from revealing any information about their values to too many processors.

In general, for any $t, 0 \leq t \leq \log p - 1$, we will define a partial assignment α that fixes the input at all but $\lfloor (p-1)/2^t \rfloor$ free processors. On any input $\boldsymbol{x}$ consistent with the partial assignment α, the input x_i of a free processor i will be known only to a set of contiguous processors containing i at time t.

We now state and prove the main lemma leading to the proof of Theorem 3.2.

LEMMA 3.3. *Fix an algorithm A. For any $t, 0 \leq t \leq \log p - 1$, there is a partial assignment α with at least $\lfloor (p-1)/2^t \rfloor$ free processors with the following property. On any input $\boldsymbol{x}$ consistent with α, the input x_i of a free processor i will be known only to a set of contiguous processors S_i containing i at time t, where $0 \notin S_i$. Moreover, for any two distinct free processors i and j, $S_i \cap S_j = 0$.*

Proof. The proof is by induction on t. The base case is when $t = 0$. In this case, since no communication has taken place, it is immediate that if α is the partial assignment that is not fixed anywhere, then all possible inputs $\boldsymbol{x}$ are consistent with α, all processors in the range $1, \ldots, p-1$ are free, and the value x_i of every processor i is known only to processors in the set $S_i = \{i\}$.

Suppose the lemma is true for $t - 1 \geq 0$, and let α be the partial assignment as in the statement of the lemma. Suppose that at round t, active processors broadcast to the right (the other case, when active processors broadcast to the left, is handled similarly).

We will define a partial assignment α' which extends α and satisfies the lemma for time t. To do this, we consider the free processors at time $t - 1$ in order from that with the largest index to that with the smallest index. (We consider processors in the opposite order in the case that the broadcast is to the left.) If free processors j, i occur consecutively in this ordering, with $j > i$, we say that j is i's *free neighbor to the right* at time $t - 1$.

For each of these processors i in turn, we will determine whether i remains free at round t, and if not, we will extend α to fix input value x_i. If i does remain free, we will define a corresponding set S'_i containing i, and will show that at time t, on any input $\boldsymbol{x}$ consistent with α', x_i is known only to processors in S'_i, that $0 \notin S'_i$, and that $S'_i \cap S'_j$ are disjoint, for free processors $i \neq j$.

Hence consider some processor i that is free at time $t - 1$. We say that S_i is *potentially active* if there is some input consistent with α such that some processor in S_i broadcasts at round t with that input. Otherwise S_i is said to be *inactive.*

If S_i is potentially active, then we define i to be free at round t if and only if the following conditions hold: (i) it is not the largest numbered free processor at time $t-1$, and (ii) processor i's free neighbor to the right at time $t-1$ is not free at time t. (Note that since i's free neighbor to the right has index $j > i$, and since we consider the free processors in order from the largest to the smallest, it is already determined whether j is free at time t.) The corresponding set S_i' is defined to be the smallest contiguous set containing S_i and S_j, where j is i's free neighbor to the right at time $t-1$. Otherwise, i is not free at time t and the value of x_i is fixed in α', to force some processor in S_i to be active at round t. It is important to note that since the processors in S_i do not know the values of x_j for the free processors $j \neq i$ at time $t-1$, then some assignment to the input x_i will force some processor in S_i to be active at round t regardless of any assignment to other inputs whose processors are free at time $t-1$.

If S_i inactive, then we define i to be free at round t and the corresponding set of processors S_i' to be equal to S_i. This completes the description of α' and the set S_i' for each free processor i.

We now argue that α' satisfies the lemma at time t. It is straightforward to see from the construction that for each free processor i at time t, $i \in S_i'$ and S_i' is a set of contiguous processors. Also, for any two distinct free processors, i and j, $S_i' \cap S_j' = 0$. This is because the S_i are contiguous, nonoverlapping sets, and each S_i' is either S_i or is formed by "collapsing" two neighboring sets S_i and S_j, where processor j is free at time $t-1$ but not at time t. Finally, using the fact that no set S_i contains processor 0, we show that no set S_i' contains processor 0. This is easy to see if S_i' equals S_i, since we know S_i does not contain processor 0. Otherwise, S_i' is the smallest contiguous set containing S_i and S_j, where j is i's free neighbor to the right at time $t-1$. Since $0 < i < j$, processor 0 cannot lie between the contiguous sets S_i and S_j. This, together with the fact that neither S_i nor S_j contain processor 0, implies that S_i' does not contain processor 0.

We next show that for any $\boldsymbol{x}$ consistent with α', if i is free at time t then x_i is known only to those processors in S_i'. First note that since processor 0 is not in S_i for any processor i that is free at time $t-1$, the instruction broadcast by 0 at time t does not reveal any information about the values of processors that are free at time $t-1$. Also, it is clear that if S_i is inactive at time t, for all $\boldsymbol{x}$ consistent with α', then x_i is still known only to those processors in S_i at time t.

Consider the other case, where S_i is potentially active at time t. Then i's free neighbor to the right at time $t-1$, say processor j, is free at time $t-1$ but not at time t. Moreover, by our construction of α', on any $\boldsymbol{x}$ consistent with α' there is a processor b in S_j that broadcasts at time t. Hence, on any input $\boldsymbol{x}$ consistent with α', any broadcast of a processor in set S_i reaches only processors in the segment between this active processor and processor b. The processors in this segment are contained in the smallest contiguous set containing both S_i and S_j. Hence, x_i is known only to processors in S_i' at time t.

To complete the proof, it remains to show that there are $\geq \lfloor (p-1)/2^t \rfloor$ free processors at time t. By the inductive hypothesis, there are $\geq \lfloor (p-1)/2^{t-1} \rfloor$ free processors at time $t-1$. If i and j are two neighboring free processors at time $t-1$, then at least one of these is still free at time t. To see this, suppose that $i < j$ and that j is not free at time t. Then either S_i is inactive at time t, in which case i is free at time t, or S_i is potentially active, in which case both conditions (i) and (ii) are satisfied, so again i is free at time t. Hence the number of free processors at time t is at least $\lfloor \lfloor (p-1)/2^{t-1} \rfloor /2 \rfloor = \lfloor (p-1)/2^t \rfloor$, as required. □

The proof of Theorem 3.2 now follows easily from Lemma 3.3. If $p \geq 2$ then $\lfloor (p-1)/2^{\log p - 1} \rfloor \geq 1$. Hence by the lemma, there is a partial assignment α that is not fixed at one free processor, say i, with the following property. On any input $\boldsymbol{x}$ consistent with α, at time $\log p - 1$, the input x_i will be known only to a set of processors S_i, where $0 \notin S_i$. Hence, x_i is unknown to processor 0 at time $\log p - 1$.

Lower bounds of $\log p$ time for PARITY and MINIMUM follow immediately from Theorem 3.2, as discussed after the statement of that theorem. The same lower bound must also hold for SUM, since PARITY can be computed from SUM without any communication. Thus, we have the following theorem.

THEOREM 3.4. *On a one-dimensional sub-bus mesh with p processors, the time to compute* PARITY, SUM, *and* MINIMUM *is at least* $\log p$.

In order to obtain lower bounds for MAJORITY and many other symmetric Boolean functions we need to modify Lemma 3.3. If α is a partial assignment, define α_0 to be the number of inputs fixed to 0 in α and α_1 to be the number of inputs fixed to 1 in α. We say a partial assignment α is *b-balanced* if $0 \leq \alpha_b - \alpha_{1-b} \leq 1$. That is, α is 1-balanced if the number of inputs assigned to 1 in α is equal to or one greater than the number of inputs assigned to 0 in α. Similarly, α is 0-balanced if the number of inputs assigned to 0 in α is equal to or one greater than the number of inputs assigned to 1 in α.

LEMMA 3.5. *Fix an algorithm A. For any bit b and for any* $t, 0 \leq t \leq \log_3 p - 1$, *there is a b-balanced partial assignment* α *with at least* $\lfloor (p-1)/3^t \rfloor$ *free processors with the following property. On any input* $\boldsymbol{x}$ *consistent with* α, *the input* x_i *of a free processor i will be known only to a set of contiguous processors* S_i *containing i at time t, where* $0 \notin S_i$. *Moreover, for any two distinct free processors i and j,* $S_i \cap S_j = 0$.

Proof. This proof is similar to that of Lemma 3.3. Assume we have a b-balanced partial assignment α at time $t-1$ and a number n of free processors with their associated segments satisfying the condition of the lemma. Assume also that at time t there is a broadcast to the right. As in the proof of Lemma 3.3, a segment is inactive if no processor in the segment would become active on any input consistent with α and is potentially active otherwise. As before, any processor i that is free at time $t-1$ and whose segment S_i is inactive remains free at time t. Assume the free processors at time $t-1$ are indexed by $i_1, i_2, \ldots, i_n$, where $i_j > i_{j+1}$ for $1 \leq j \leq n$. We consider these processors three at a time, largest index to smallest, to determine which potentially active processors remain free at time t and for those that do not remain free, what their inputs will be assigned to in the new b-balanced partial assignment α'. If n is divisible by 3 then this process will end simply. If not, there will be a remaining group of 1 or 2 that must be dealt with.

Assume that for $j \leq 3m$, it has already been determined whether i_j is free at time t and if not, what the assignment to x_{i_j} is in the assignment α'. We now consider the processors i_{3m+1}, i_{3m+2}, and i_{3m+3}, where $3m+3 \leq n$. There are four cases to consider, depending on how many of the segments $S_{i_{3m+1}}, S_{i_{3m+2}}, S_{i_{3m+3}}$ are potentially active. If none are potentially active then there is nothing to do. If exactly one, say $S_{i_{3m+k}}$, is potentially active, then set $x_{i_{3m+k}}$ to a value to make α' b-balanced. That is, if $\alpha'_0 = \alpha'_1$ then assign $x_{i_{3m+k}}$ to b, otherwise set it $1-b$. If exactly two of the segments are potentially active, then assign the input associated with one to 0 and the other to 1. Finally, if all three are potentially active then i_{3m+3} remains free, $x_{i_{3m+2}}$ is set so as force a processor in the segment $S_{i_{3m+2}}$ to be active, and then $x_{i_{3m+1}}$ is set to make α' b-balanced.

Once the groups of three have been processed there may be one or two remaining free processors. If exactly one of the segments in this remaining group is potentially active, then assign the input of that processor to a value to make the partial assignment b-balanced. If exactly two of the segments of the free processors in this remaining group are potentially active, then assign the two inputs of the free processors to opposite values.

At the end of this process, at least $\lfloor n/3 \rfloor$ of the processors are free. If i is free at time t and S_i is inactive, then $S_i' = S_i$. If i is free at time t and S_i is potentially active, then the corresponding set S_i' is defined to be the smallest contiguous set containing S_i and S_j, where j is i's free neighbor to the right at time $t-1$. This happens in the fourth case above when $i = i_{3m+3}$ and $j = i_{3m+2}$.

It should be clear that α' is a b-balanced partial assignment with at least $\lfloor (p-1)/3^t \rfloor$ unassigned inputs. Furthermore, for the same reasons as in the proof of Lemma 3.3, for any input $\boldsymbol{x}$ consistent with α', the input x_i of a free processor i will be known only to a set of contiguous processors S_i containing i at time t, where $0 \notin S_i$. Clearly, the segments at time t are disjoint. □

As a consequence of Lemma 3.5 we have the following theorem, which allows us to find a b-balanced input with a component unknown to processor 0 at a time slightly less than the maximum time to find just some input with a component unknown to processor 0, as in Theorem 3.2.

THEOREM 3.6. *On a one-dimensional sub-bus mesh with p processors and for any algorithm A and bit b, there exists a b-balanced input $\boldsymbol{x}$ such that for some $i, 1 \leq i < p-1$, x_i is set to b, but is unknown to processor* 0 *at time* $\log_3 p - 1$.

Proof. If $p \geq 2$ then $\lfloor (p-1)/3^{\log_3 p - 1} \rfloor \geq 1$. By Lemma 3.5, there is a $(1-b)$-balanced partial assignment α that is not fixed at a free processor i. Set $x_i = b$, then set the remaining unassigned inputs so as to make the input b-balanced. Since 0 is not a member of the segment S_i at time $\log_3 p - 1$, x_i is not known to processor 0. □

THEOREM 3.7. *On a one-dimensional sub-bus mesh with p processors, the time to compute* MAJORITY *is at least* $\log_3 p$.

Proof. Let A be any algorithm for MAJORITY. There are two cases to consider, depending on whether p is even or odd. If p is even then by Theorem 3.6 select a 0-balanced input $\boldsymbol{x}$ and an i such that $x_i = 0$ and x_i is not known to processor 0 at time $\log_3 p - 1$. Clearly, processor 0 cannot have computed the majority of the inputs by time $\log_3 p - 1$ since its computation would be identical for the input $\boldsymbol{x}$ and $\boldsymbol{x}'$, which is identical to input $\boldsymbol{x}$ except that $x_i' = 1$. The latter input has a majority of 1's while the former does not. If p is odd then select a 1-balanced input $\boldsymbol{x}$ and an i such that $x_i = 1$ and x_i is not known to processor 0 at time $\log_3 p - 1$. The remainder of the argument is similar to that above. □

For any symmetric Boolean function f on p inputs define $m(f)$ to be the k such that $\frac{p}{2} - k \geq 0$ is minimal and for some bit b the value of f on an input with exactly k inputs equal to b differs from the value of f on an input with $k+1$ inputs equal to b. For example, $m(\text{MAJORITY}) = m(\text{PARITY}) = \lfloor \frac{p}{2} \rfloor$ and $m(\text{OR}) = m(\text{AND}) = 0$.

COROLLARY 3.8. *On a one-dimensional sub-bus mesh with p processors, the time to compute any symmetric function f is at least* $\log_3(2m(f))$.

Proof. Let f be given. Let b be such that if $m(f)$ inputs have the value b then f has one value and if $m(f)+1$ inputs have the value b then f has another value. For simplicity consider the case in which $m(f)$ inputs equal 0 implies the value of f is 0 and $m(f)+1$ inputs equal 0 implies the value of f is 1. If exactly $p - 2m(f)$ inputs are set to 0 then the restricted function has $2m(f)$ inputs. By a proof identical to the proof of Theorem 3.7, any algorithm to compute the restricted function must take time $\log_3(2m(f))$. The argument for $b = 1$ is similar. □

Define THRESHOLD$_k$ to be the Boolean function that is 0 with k or fewer inputs set to 1 and 1 otherwise. Clearly, $m(\text{THRESHOLD}_k) = \min(k, p-k)$. Thus, we have the following corollary.

COROLLARY 3.9. *On a one-dimensional sub-bus mesh with p processors, the time to compute* THRESHOLD$_k$ *is at least* $\log_3(2\min(k, p-k))$.

4. Algorithms for PARITY and SUM.

In this section we present asymptotically optimal algorithms for PARITY and SUM on the two-dimensional sub-bus mesh computer. We start with the PARITY algorithm. It is the simpler of the two, and introduces some of the key ideas that are useful in the SUM algorithm.

4.1. PARITY algorithm.

We will introduce a series of problems in increasing order of difficulty. The algorithm for each problem will lead to the next one with some fresh tricks. This will help us concentrate on one idea at a time.

Each of the algorithms below can be executed on a submesh of the $\sqrt{p} \times \sqrt{p}$ mesh. By an *array* or *subarray* we mean a submesh of the full mesh which may be nonsquare and noncontiguous. In case it is noncontiguous, it is assumed that the processors between any two processors in the subarray are inactive so as not to interfere with communication between the processors in the subarray. Furthermore, any of the algorithms below can be executed in parallel on disjoint and properly aligned subarrays of the full $\sqrt{p} \times \sqrt{p}$ array. If the algorithm is executed on a $m \times n$ subarray, then we say processor (i, j) is the processor in the (i, j)th position (the ith column and jth row) of the subarray, where $0 \le i < m$ and $0 \le j < n$. Although it is not generally the case that processor (i, j) has its PIDx $= i$ and PIDy $= j$, it will always be the case that i, j, and dimensions of the subarray can be computed from the PID of the processor and other local data in constant time.

LEMMA 4.1. *On an $n \times 2^n$ array with each processor in the top row having an input bit, the parity of the input bits can be computed in constant time.*

Proof. There are 2^n possible inputs, so we will make row j of the array responsible for determining whether the input, thought of as an integer x with $0 \le x < 2^n - 1$, actually equals j. In particular, processor (i, j) determines if the input in processor $(i, 0)$ equals the ith bit of j. A downward broadcast of the input gives processor (i, j) knowledge of the input in processor $(i, 0)$. Then processor (i, j) compares the input of processor $(i, 0)$ with the ith bit of j. A constant time AND of the outcomes of these comparisons in all the rows in parallel tells processor $(0, j)$ whether the input, thought of as an n-bit number, equals j. This information can then be broadcast up to processor $(0, 0)$. Since $2^n \le \sqrt{p}$, we know $j \le \log p$ so that processor $(0, 0)$ can compute the sum of the bits in j in constant time, using the fact that computing the sum of the bits of an input is in uniform NC. The parity of the input bits is the parity of this sum. □

We saw that with exponentially many rows we can compute the parity in constant time. In general, if we have more than a constant number of rows, we can beat the straightforward $O(\log n)$ time algorithm.

LEMMA 4.2. *On an $n \times m$ array with each processor in the top row having an input bit, the parity of the input bits can be computed in time $O(\frac{\log n}{\log\log m})$.*

Proof. We can think of the $n \times m$ array as $\frac{n}{\log m}$ subarrays of dimension $\log m \times m$ placed side by side. As in the previous proof, we can compute the parity of groups of $\log m$ bits in constant time. This leaves $\frac{n}{\log m}$ partial results in the first row of an array of dimension $\frac{n}{\log m} \times m$. Repeating the process $\frac{\log n}{\log\log m}$ times we have the parity of all the n bits. □

So far we have been assuming that only the processors in the top row have inputs. Let us now consider the case where each processor has an input.

LEMMA 4.3. *On an $n \times m$ array with each processor having an input bit, the parity of the input bits can be computed in time $O(\log m + \frac{\log n}{\log\log m})$.*

Proof. First, in parallel, the processors within each column run the one-dimensional PARITY algorithm described in §3.1. This part takes time $O(\log m)$. At this point, we have partial results stored in the top row. From the previous lemma, the parity of these partial results can be computed in an additional $O(\frac{\log n}{\log\log m})$ steps. □

We are ready to give our PARITY algorithm.

THEOREM 4.4. *On a $\sqrt{p} \times \sqrt{p}$ mesh with each processor having an input bit,* PARITY *can be computed in time $O(\frac{\log p}{\log\log p})$.*

Proof. Think of the $\sqrt{p} \times \sqrt{p}$ mesh as $\frac{\sqrt{p}}{m}$ smaller arrays of dimension $\sqrt{p} \times m$, one on top of the other. Each of these arrays computes the parity of its input bits in parallel. By the previous lemma, this takes $O(\log m + \frac{\log\sqrt{p}}{\log\log m})$ time and leaves $\frac{\sqrt{p}}{m}$ partial results in the leftmost column. By Lemma 4.2 their parity can be computed in time $O(\frac{\log\sqrt{p}}{\log\log(\sqrt{p}/m)})$. Choosing $\log m = \frac{\log p}{\log\log p}$ we get a total running time of $O(\frac{\log p}{\log\log p})$. □

4.2. SUM algorithm. Computing PARITY is the same as computing the sum of the inputs modulo 2. Lemmas 4.1 and 4.2 can be generalized to compute the sum, modulo a small integer, of inputs on the top row. For all the problems below we assume that the inputs are nonnegative integers of length $O(\log p)$.

LEMMA 4.5. *If $\log Q \leq \sqrt{\log n}$ then on an $n \times n$ array with each processor having Q and with each processor in the top row having an input integer, the sum of the inputs modulo Q can be computed in time $O(\frac{\log n}{\log\log n})$.*

Proof. Let $m = \frac{\log n}{\log Q}$. Let us focus on an $m \times n$ subarray that has m inputs on the top row. There are $Q^m = n$ possibilities for the m inputs mod Q. For $0 \leq j < n$, think of j as an integer written in base Q. As in the computation of parity, processor (i, j) is responsible for determining if the ith input mod Q is equal to the ith Q-ary digit of j. Processor (i, j) learns of the input at $(i, 0)$ by a broadcast down from the first row. Then, processor $(0, j)$ learns from an AND on its row that the ith Q-ary digit of j is the ith input mod Q for all i such that $0 \leq i < m$. Processor $(0, j)$ then transmits j to processor $(0, 0)$ where the sum mod Q of the Q-ary digits of j is computed.

The original $n \times n$ array can be divided into $\frac{n}{m}$ subarrays of dimension $m \times n$ placed side by side where the algorithm above is performed in parallel. What remains are $\frac{n}{m}$ numbers in the top row. Iterate this process $O(\frac{\log n}{\log m}) = O(\frac{\log n}{\log\log n - \log\log Q})$ times to compute the sum of all the n integers modulo Q.

To complete the proof we must argue that computing the sum mod Q of the Q-ary digits of a number x is in uniform NC. We assume both Q and x are written in binary. First, since $\log Q \leq \sqrt{\log n}$ and $n \leq \sqrt{p}$, then the length of Q is $O(\sqrt{\log p})$. Second, $x \leq \sqrt{p}$, so that x is of length $O(\sqrt{\log p})$. Thus, the lengths of Q and x can be assumed to be bounded by the same number b, which we can assume is a power of two and of length $O(\sqrt{\log p})$. To find the sum mod Q of the Q-ary digits of x, write x as $x_0 + Q^{b/2}x_1$, by dividing x by $Q^{b/2}$. Recursively, find the a_0 and a_1 that are the sums mod Q of the Q-ary digits of x_0 and x_1, respectively. Then, $a = (a_0 + a_1) \bmod Q$ is the sum mod Q of the Q-ary digits of x. The necessary powers of Q, division by these powers, and sum are all in uniform NC. Since the number of levels of recursion is bounded by $\log_2 b$, then the result a can also be computed in uniform NC. □

By the Chinese remainder theorem we know that if we can compute the sum modulo sufficiently many small integers, we can compute the exact sum.

LEMMA 4.6. *If $6 \leq \log t \leq \sqrt{\log n}$ then on an $n \times tn$ array with each processor in the top row having an input integer such that the sum of these integers is less than*

2^t, *the sum of the inputs can be computed in time* $O(\frac{\log n}{\log\log n})$.

Proof. Think of the $n \times tn$ array as t subarrays, each of size $n \times n$, one on top of the other. Let M be the product of all primes less than t. If $t \geq 41$ then $M \geq 2^t$ [27, Cor. to Thm. 4, p. 70]. Hence, if the sum of the input integers is less than 2^t then it is enough to compute the sum modulo M. We already know how to compute the sum modulo small primes. Our plan is to let each $n \times n$ subarray compute the sum modulo a different prime and then apply the Chinese remainder algorithm to compute the sum modulo M.

To begin with, the processors in the top row broadcast the input values down the columns. The jth subarray decides whether j is a prime. This can be done in two stages. A number j is prime if and only if it is not divisible by any number between 1 and $\sqrt{j}$. In the first stage, assign $\sqrt{j}$ processors in the first row to check for each possible divisor. In the second stage, these processors compute an AND of their results. Only processors in the jth subarray for prime j participate in all subsequent steps. The jth subarray computes a_j, the sum of all the inputs modulo j. By Lemma 4.5, this can be done in $O(\frac{\log n}{\log\log n})$ time.

Next, in $O(\log t)$ steps, each processor in the jth subarray computes M, the product of all primes less than t, and $m_j = \frac{M}{j}$. The processors in the jth subarray compute $(a_j m_j)((m_j)^{-1} \bmod j)$. This can be done in constant time. The nontrivial part is computing $((m_j)^{-1} \bmod j)$. There are at most j possible values for the inverse. We assign j processors in (say) the first row of the jth subarray for each possible value of the inverse. In one step, each of these assigned processors can check whether it has the right value of the inverse. The processor corresponding to the right value of the inverse broadcasts this to all other processors. By the Chinese remainder theorem, the exact value of the sum is the summation of $(a_j m_j)((m_j)^{-1} \bmod j)$, which can be computed in $O(\log t)$ steps.

In total the computation can be done in $O(\log t + \frac{\log n}{\log\log n}) = O(\frac{\log n}{\log\log n})$ time. □

LEMMA 4.7. *If* $tw \leq m$ *and* $6 \leq \log t \leq \sqrt{\log w}$ *then on an* $n \times m$ *array with each processor in the top row having an input integer such that the sum of these integers is less than* 2^t, *the sum of the inputs can be computed in time* $O(\frac{\log n}{\log\log w})$.

Proof. Think of the $n \times m$ array as $\frac{n}{w}$ subarrays, each of dimension $w \times m$, side by side. Since $tw \leq m$, each $w \times m$ subarray can be thought of as containing a $w \times tw$ subarray at the top. Each $w \times tw$ subarray has its input on the top row. By Lemma 4.6, the sum of the inputs of all the $w \times tw$ subarrays can be computed in time $O(\frac{\log w}{\log\log w})$ leaving $\frac{n}{w}$ partial sums in the top row. This process is repeated $\frac{\log n}{\log w}$ times until the input is reduced to a single number. This reduction takes time $O(\frac{\log n}{\log\log w})$. □

We now consider the case where *each* processor has an input.

LEMMA 4.8. *If* $tw \leq m$ *and* $6 \leq \log t \leq \sqrt{\log w}$ *then on an* $n \times m$ *array with each processor having an input integer such that the sum of these integers is less than* 2^t, *the sum of the inputs can be computed in time* $O(\log m + \frac{\log n}{\log\log w})$.

Proof. The first step is simply to add the columns in parallel in time $O(\log m)$. We are now reduced to the problem in the previous lemma. Hence the total time is $O(\log m + \frac{\log n}{\log\log w})$. □

THEOREM 4.9. *On a* $\sqrt{p} \times \sqrt{p}$ *mesh with each processor having an input integer of length* $O(\log p)$, SUM *can be computed in time* $O(\frac{\log p}{\log\log p})$.

Proof. Choose $t = \log(p^k)$, where p^k is an upper bound on the sum of the input integers and $\log t \geq 6$. Let c be a constant, which depends only on k, such that $\log t \leq \sqrt{\frac{c \log p}{\log \log p}}$. Now, choose w such that $\log w = \frac{c \log p}{\log \log p}$. Let $m = tw$, so that t, w, and m satisfy the hypothesis of Lemma 4.8. Think of the $\sqrt{p} \times \sqrt{p}$ mesh as $\frac{\sqrt{p}}{m}$ arrays of dimension $\sqrt{p} \times m$ one on top of the other. By Lemma 4.8 the sum of these arrays can be computed in time $O(\log m + \frac{\log \sqrt{p}}{\log \log w})$, leaving $\frac{\sqrt{p}}{m}$ partial sums in the first column. We may now apply Lemma 4.7 to these inputs to find the full sum in an additional $O(\frac{\log \sqrt{p}}{\log \log w})$ time. The total time is then $O(\log m + \frac{\log \sqrt{p}}{\log \log w})$. Since $\log w = \frac{c \log p}{\log \log p}$ and $\log m = \log w + \log t = O(\frac{\log p}{\log \log p})$, the total time of the algorithm is $O(\frac{\log p}{\log \log p})$. □

It is interesting to note that if we assume that the individual processors can operate on integers of arbitrary length in constant time, then using the technique of Theorem 4.9, the sum of p integers of length $2^{O(\sqrt{\log p / \log \log p})}$ can be computed in time $O(\frac{\log p}{\log \log p})$.

Since MAJORITY can be computed from SUM in constant time we may now state the following theorem.

THEOREM 4.10. *On a two-dimensional sub-bus mesh with p processors,* PARITY, MAJORITY, *and* SUM *can be computed in time* $O(\frac{\log p}{\log \log p})$.

In the next section we will show these bounds are optimal.

5. Simulation of a sub-bus mesh by a CRCW PRAM. In this section, we prove a $\Omega(\frac{\log p}{\log \log p})$ lower bound for computing PARITY, MAJORITY, and SUM on the two-dimensional sub-bus mesh computer. To prove this, we show that any algorithm for the sub-bus model can be simulated by a CRCW PRAM algorithm with only a constant-factor loss in running time. We then apply lower-bound results for PARITY on the PRAM model. We begin this section by describing the PRAM results. Then we describe our lower-bound model and describe the simulation in detail.

Beame and Håstad [4] considered lower bounds for the following "ideal" CRCW PRAM model. There are $p(n)$ numbered processors that share $c(n)$ numbered memory cells, where $p(n)$ and $c(n)$ are polynomially bounded. There is no bound on the possible contents of a memory cell. Initially, the input bits $x_0, \ldots, x_{n-1}$ are stored in the first n memory cells and the remaining cells have value 0. Before each step t, a processor is in some state, say q. At the tth step, the processor may read the value v stored in some memory cell C. Based on C, v, and q, the processor moves to a new state q', and may write a value v' to some cell C'. There is no limit on the number of states of a processor nor on the resources needed to compute v' and C'. If several processors attempt to write into the same memory cell at the same step, the lowest numbered processor succeeds. This model is called the *priority* CRCW PRAM. Beame and Håstad [4] have shown that the time to compute any of PARITY, MAJORITY, and SUM on the ideal CRCW PRAM is $\Omega(\log n / \log \log n)$.

In our lower bound model of the two-dimensional sub-bus there are p processors, numbered $0, 1, \ldots, p-1$, connected in a $\sqrt{p} \times \sqrt{p}$ mesh as in the two-dimensional upper bound model. Processor 0 is the front-end. The computation proceeds in rounds costing one unit, and in each round, the processors first communicate and then perform arbitrary internal computation. The communication is done just as in the upper-bound model. Again, there is no bound on the size of values broadcast or computed. The main result of this section is the following theorem.

THEOREM 5.1. *Any problem that can be solved in time* $t(p)$ *on a two-dimensional* *sub-bus mesh computer with p processors can be solved in time $O(t(p))$ on a priority CRCW PRAM with $O(p^{3/2})$ processors.*

Proof. Let A be a $T(p)$-time algorithm for solving a problem on the two-dimensional sub-bus lower bound model. We describe an ideal CRCW PRAM that simulates A, such that each round of A takes $O(1)$ steps.

In the simulating PRAM, there are p processors, numbered $0, 1, \dots, p-1$, corresponding to the p processors of the sub-bus model. There are also auxiliary processors, whose computation will be described later.

There are two special memory cells, called Condition and Statement, used by processor 0 to communicate the instruction at each round. Also, corresponding to each processor $i, 0 \leq i \leq p-1$, there are the following special memory cells (we do not specify their exact addresses, but assume they are computable by processor i). Active(i) is used to denote at each round whether i is active. It is initialized to false at the beginning of each round. Send(i) is used to store the value broadcast by i, at each round, if i is active. Receive(i) is used to store the value, if any, received by i in each round. At the start of each round, processor i sets Active(i) to false and sets Receive(i) to some special value which is not in the range of possible values that can be broadcast by A.

We now describe the simulation of a single round of A. First, processor 0 writes the strings <*condition*> and <*statement*> in cells Condition and Statement, respectively. Each processor $i, 0 \leq i \leq p-1$ reads these cells and decides if it is active at this round. If so, i writes the value to be broadcast in Send(i) and sets Active(i) to true.

We next describe how each processor i determines the value it receives (if any) during the broadcast instruction. If i receives a value, it is from one of $\sqrt{p}$ processors on either the vertical or horizontal bus along which i is connected. Let these processors be numbered $i_1, \dots, i_{\sqrt{p}}$, where the ordering is such that if i_1 is active, then i_1 is the processor from which i receives a message; if i_1 is not active but i_2 is, then i_2 is the processor from which i receives a message, and so on, so that if i_k is active and none of $i_1, \dots, i_{k-1}$ are active, then i_k is the processor from which i receives a message. For example, if the direction of communication is up, then the sequence is $(i + \sqrt{p}) \bmod p, (i + 2\sqrt{p}) \bmod p, \dots, (i + \sqrt{p}\sqrt{p}) \bmod p$.

Each processor i has $\sqrt{p}$ auxiliary processors to help it compute the value it receives, if any. Let the auxiliary processors of i be numbered $n_i + 1, \dots, n_i + \sqrt{p}$, where $n_i = p + i\sqrt{p} - 1$. Each processor $n_i + k$ computes i_k; this can be done by reading <*statement*>, to determine the direction of communication. If processor i_k is active, that is, Active(i_k) is true, then processor $n_i + k$ reads the value Send(i_k) and writes it in Receive(i). Because of the ordering of the auxiliary processors, and the priority write conflict resolution assumption, the value written in Receive(i) is the value received by processor i at that round of A, in the sub-bus computation.

Once the cells Receive(i) have been computed, each processor $i, 0 \leq i \leq p-1$ completes the round by simulating the internal computation of the ith sub-bus processor. This completes the description of the simulation. It is clear that all the steps described can be done by the processors of the ideal CRCW PRAM, since they have unbounded resources with which to compute at each step, and an unbounded number of states that can be used to store the internal configurations of the processors of the sub-bus mesh computer. □

As a direct consequence of Theorem 5.1 we have the following theorem.

THEOREM 5.2. *On a two-dimensional sub-bus mesh with p processors, the time to compute* PARITY, MAJORITY, *and* SUM *is* $\Omega(\frac{\log p}{\log \log p})$.

6. MINIMUM. In this section we survey the two-dimensional complexity of MINIMUM in the comparison model of the sub-bus mesh. Previous work for the reconfigurable mesh shows that the time to compute MINIMUM on the two-dimensional mesh is $\Theta(\log\log p)$. For completeness, we include the algorithm, adapted to the sub-bus model.

THEOREM 6.1 (see [21, 31]). *In the comparison model of a two-dimensional sub-bus mesh computer with p processors, the time to compute* MINIMUM *is* $\Theta(\log\log p)$.

Proof. Recall that in the comparison model we assume the only operations allowed on input values are comparison, copy, and broadcast. With this restriction any sub-bus mesh algorithm for MINIMUM can be thought of as a "parallel comparison tree" as defined by Valiant [31]. In this model, any p comparisons of arbitrary input values can be made in one step. Depending on the outcome of these comparisons, one of 2^p branches to the next step can be made. Valiant showed that in the parallel comparison tree model with p processors, $\Omega(\log\log p)$ steps are necessary to determine the minimum of p inputs.

In the same paper Valiant [31], gave an algorithm for the minimum which runs in $O(\log\log p)$ time. The Valiant algorithm can be realized on the reconfigurable mesh computer as shown by Miller et al. [21]. In fact, their work also applies to the two-dimensional sub-bus mesh computer. In the Valiant algorithm, assume there are p processors, where $p = 2^{2^k}$ for some k. With one parallel comparison the number of possible minima to consider is reduced to $\frac{p}{2}$. Subsequently, if there are $\frac{p}{b}$ possible minima remaining, then in one parallel comparison this number can be reduced to $\frac{p}{b^2}$. This is done by dividing the $\frac{p}{b}$ numbers into $\frac{p}{b^2}$ groups of size b and using one parallel comparison to find the minimum of all the groups simultaneously. A group of size b requires b^2 processors to find the minimum in one parallel comparison. Thus, p processors are utilized to find the $\frac{p}{b^2}$ possible minima.

The Valiant algorithm can be realized on the two-dimensional sub-bus mesh computer. The basic building block is the MINIMUM algorithm described in §2.2, which in constant time finds the minimum of n values if they are in the first row of an $n \times n$ array of processors. In the two-dimensional algorithm, if there are $\frac{p}{b}$ values remaining (for $b \geq 2$) then there are $\frac{p}{b^2}$ subarrays of the $\sqrt{p} \times \sqrt{p}$ array, each of which is $b \times b$ with b inputs in the first row of the subarray. The leftmost corner of the subarrays appear at processors (ib, jb), where $0 \leq i, j < \frac{\sqrt{p}}{b}$. Using the basic building block, the $\frac{p}{b}$ possible minima can be reduced to $\frac{p}{b^2}$ possible minima in constant time. These possible minima are located at processors indexed (ib, jb), where $0 \leq i, j < \frac{\sqrt{p}}{b}$. By first broadcasting these values to the right, then selectively broadcasting these values up, we end up with $\frac{p}{b^2}$ remaining values in the first row of $\frac{p}{b^2}$ subarrays, each of size $\frac{p}{b^2} \times \frac{p}{b^2}$. The iterative version of this algorithm is given below.

```
MINIMUM of $p$ values on a two-dimensional mesh
input: plural number variable x
output: MINIMUM value $x_{i,j}$
other: plural number variable y, Boolean variable t, singular integer b
NOTE: We assume $p = 2^{2^k}$ for some $k$.
begin
  b ← 2
  broadcast_up[1].y ← x
  if y < x then
    x ← y
  repeat begin
```

```
        Step 1
        if PIDy mod b = 0 then
           broadcast_down[b-1].x ← x
        if PIDx mod b = PIDy mod b then begin
           y ← x
           broadcast_left[b-1].y ← y
        endif
        if PIDx mod b = 0 then
           broadcast_right[b-1].y ← y
        t ← (x > y)
        if t then
           broadcast_up [b-1].t ← true
        if not t then
           broadcast_left[b-1].x ← x
        Step 2
        if b = $p$ then
           return $x_{0,0}$
        if PIDy mod b = 0 then begin
           if PIDx mod b = 0 then
              broadcast_right[b-1].x ← x
           if PIDx mod b = (PIDy mod $b^2$) / b then
              broadcast_up[$b^2$-1].x ← x
        endif
        b ← $b^2$
     endrepeat
end
```

To explain the algorithm in more detail, at the beginning of step 1 there are possible minima in every processor along every bth row. Then step 1 does the constant time MINIMUM on each $b \times b$ block in parallel. Within each block, it distributes the b initial values into the variables x and y for each processor. Thus, the processor in position (i, j) in each block has the initial value of processor $(i, 0)$ in its x register and the initial value of processor $(j, 0)$ in its y register. A comparison of x and y is recorded in the Boolean value t. After broadcasting t up the only processor with the value false is the processor in position $(i, 0)$ which has the minimum for this block. So, that processor will broadcast its initial value x to processor $(0, 0)$ within the block.

In step 2 we have the situation where the processors (ib, jb) have possible minima, and we want to move them all to rows, so that the processors (i, jb^2) all have potential minima. This is accomplished in two substeps. First, broadcast the potential minima to the right. Second, selectively broadcast the minima up. That is, each potential minimum at processor $(ib + k, jb^2 + kb)$ for $0 \leq k < b$ is broadcast up.

In case p is not of the form 2^{2^k} for some k then the algorithm must be modified slightly. In the modified algorithm we will always maintain an active rectangular sub-mesh which is $\sqrt{p} \times q$, where $q \leq \sqrt{p}$ and b divides q evenly. In attempting step 2 it may happen that b^2 does not divide r evenly. If this is the case we set $q' = b^2 \lfloor q/b^2 \rfloor$ and use the upper $\sqrt{p} \times q'$ subarray. The blocks below this new rectangle can easily be merged into the blocks directly above them in constant time. Thus, in constant time we go from $b \times b$ blocks to $b^2 \times b^2$ blocks. This is sufficient to solve the problem in time $O(\log \log p)$. □

7. Conclusions. We have proved tight bounds (to within constant factors) on the time needed to compute several functions on the sub-bus mesh computer. For some of these problems, such as PARITY, MINIMUM, and SUM, the running times on a sub-bus mesh computer match (to within constant factors) the running times on a general PRAM. Moreover, machines based on the sub-bus mesh architecture are commercially available [5]. For these reasons, we believe that the sub-bus mesh architecture deserves further study.

Our algorithms for PARITY and SUM are probably not practical for any reasonable size p for two reasons. First the speed-up by a factor of $O(\log \log p)$ has too large a constant factor to be significant. Second, it is doubtful that hardware designers would want to implement the new NC functions required by the algorithm. It is possible to remove the second factor inhibiting practicality by adding preprocessing phases to the algorithms. A preprocessing phase uses only standard arithmetic/Boolean operations to compute values that depend only on the processor PIDs and the structure of the algorithm and which, in the original algorithm, would be computed as results of NC functions. Using preprocessing, many more values would be computed than are actually used in the algorithm, since during the preprocessing it is not known exactly which values will be needed later on. The necessary preprocessing for the two algorithms PARITY and SUM is complicated, but can be accomplished within the $O(\frac{\log p}{\log \log p})$ time bound.

We have implemented the $O(\log \log p)$ MINIMUM algorithm on a 1,024 processor MasPar MP-1. The constant factor in front of the $\log \log p$ forces the algorithm to run more slowly than the standard $O(\log p)$ algorithm. However, we believe that the ideas in the $O(\log \log p)$ MINIMUM algorithm have the potential to be used in a competitive practical algorithm for finding the minimum on commercially available meshes with more than 1,024 processors.

Several open questions are suggested by our results. Are there simpler $O(\frac{\log p}{\log \log p})$ algorithms for PARITY or SUM on the two-dimensional sub-bus mesh that may be competitive on real machines? Is it possible to improve the lower bound for MAJORITY on a one-dimensional mesh from $\log_3 p$ to $\log_2 p$? Can our lower bound for PARITY on the two-dimensional sub-bus mesh can be simplified, or improved by a constant factor, using the mesh model directly rather than translating results from the PRAM model?

Acknowledgments. We thank Eric Bach, David Barrington, and Paul Beame for several useful discussions and suggestions. We thank Tosten Suel and Phil MacKenzie for providing us with several references to previous work on the reconfigurable mesh. We thank the anonymous referees for useful suggestions and for pointing out a seminal reference to the sub-bus mesh architecture.

REFERENCES

[1] M. Ajtai, J. Komlós, and E. Szemerédi, *An $O(n \log n)$ sorting network*, in Proc. 15th Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1983, pp. 1–9.

[2] A. Bar-Noy and D. Peleg, *Square meshes are not always optimal*, IEEE Trans. Comput., 40 (1991), pp. 138–147.

[3] K. E. Batcher, *Design of a massively parallel processor*, IEEE Trans. Comput., C-29 (1980), pp. 836–840.

[4] P. Beame and J. Håstad, *Optimal bounds for decision problems on the CRCW PRAM*, J. Assoc. Comput. Mach., 36 (1989), pp. 643–670.

[5] T. BLANK, *The MasPar MP-1 architecture*, in Proc. COMPCON Spring 90—35th IEEE Computer Society International Conference, IEEE Press, Piscataway, NJ, 1990, pp. 20–24.

[6] G. E. BLELLOCH, *Vector Models for Data-Parallel Computing*, MIT Press, Cambridge, MA, 1990.

[7] R. COLE, *Parallel merge sort*, SIAM J. Comput., 17 (1988), pp. 770–785.

[8] S. A. COOK, *A taxonomy of problems with fast parallel algorithms*, Inform. and Control, 64 (1985), pp. 2–22.

[9] S. A. COOK, C. DWORK, AND R. REISCHUK, *Upper and lower time bounds for parallel random access machines without simultaneous writes*, SIAM J. Comput., 15 (1986), pp. 87–97.

[10] F. FICH, P. RADGE, AND A. WIGDERSON, *Relations between concurrent write models of parallel computation*, SIAM J. Comput., 17 (1988), pp. 606–627.

[11] F. E. FICH, F. MEYER AUF DER HEIDE, P. RAGDE, AND A. WIGDERSON, *Lower bounds for parallel random access machines with unbounded shared memory*, Adv. Comput. Res., 4 (1987), pp. 1–15.

[12] E. HAO, P. D. MACKENZIE, AND Q. F. STOUT, *Selection on the reconfigurable mesh*, in Frontiers of Massively Parallel Computing, IEEE Computer Society Press, Los Alamitos, CA, 1992, pp. 38–45.

[13] W. D. HILLIS AND G. L. STEELE, JR., *Data parallel algorithms*, Comm. Assoc. Comput. Mach., 29 (1986), pp. 1170–1183.

[14] J. JÁJÁ, *An Introduction to Parallel Algorithms*, Addison–Wesley, Reading, MA, 1992.

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

[16] T. LEIGHTON, *Tight bounds on the complexity of parallel sorting*, IEEE Trans. Comput., C-34 (1985), pp. 344–354.

[17] ———, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*, Morgan Kaufmann, San Mateo, CA, 1992.

[18] H. LI AND Q. F. STOUT, EDS., *Reconfigurable Massively Parallel Computers,* Prentice–Hall, Englewood Cliffs, NJ, 1991.

[19] P. D. MACKENZIE, *A separation between reconfigurable mesh models*, Parallel Process. Lett., 5 (1995), pp. 15–22.

[20] Y. MATIAS AND A. SCHUSTER, *On the power of the* $2 \times n$ *reconfigurable mesh*, manuscript, AT&T Bell Labs, Murray Hill, NJ, 1993.

[21] R. MILLER, V. K. PRASANNA KUMAR, D. I. REISIS, AND Q. F. STOUT, *Parallel computations on reconfigurable meshes*, IEEE Trans. Comput., 42 (1993), pp. 678–692.

[22] R. MILLER AND Q. STOUT, *Mesh computer algorithms for computational geometry*, IEEE Trans. Comput., 38 (1989), pp. 321–340.

[23] K. NAKANO, *An efficient algorithm for summing up binary values on a reconfigurable mesh*, IEICE Trans. Fund. Electron., Comm. and Comput. Sci., E77-A (1994), pp. 652–657.

[24] K. NAKANO, T. MASUZAWA, AND N. TOKURA, *A sub-logarithmic time sorting algorithm on a reconfigurable array*, IEICE Trans., E74 (1991), pp. 3894–3901.

[25] V. K. PRASANNA KUMAR AND C. S. RAGHAVENDRA, *Array processor with multiple broadcasting*, Parallel Distrib. Comput., 4 (1987), pp. 173–190.

[26] D. REISIS AND V. K. PRASANNA KUMAR, *VLSI arrays with reconfigurable buses*, in Proc. Supercomputing, 1st International Conference, Athens, Greece, 1987, Lecture Notes in Comput. Sci., 297 (1988), Springer-Verlag, Berlin, pp. 732–743.

[27] J. B. ROSSER AND L. SCHOENFELD, *Approximate formulas for some functions of prime numbers*, Illinois J. Math., 6 (1962), pp. 64–74.

[28] W. L. RUZZO, *On uniform circuit complexity*, J. Comput. System Sci., 22 (1981), pp. 365–383.

[29] Q. F. STOUT, *Meshes with multiple buses*, in Proc. 27th Annual IEEE Symposium on Foundations of Computer Science, IEEE Press, Piscataway, NJ, 1986, pp. 264–273.

[30] C. THOMPSON, *Area-time complexity for VLSI*, in Proc. of 11th Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1979, pp. 81–88.

[31] L. VALIANT, *Parallelism in comparison problems*, SIAM J. Comput., 4 (1975), pp. 348–355.

SIAM J. COMPUT.
Vol. 25, No. 3, pp. 540–561, June 1996

004

PATHWIDTH, BANDWIDTH, AND COMPLETION PROBLEMS TO PROPER INTERVAL GRAPHS WITH SMALL CLIQUES*

HAIM KAPLAN† AND RON SHAMIR‡

Abstract. We study two related problems motivated by molecular biology.

- Given a graph G and a constant k, does there exist a supergraph G' of G that is a unit interval graph and has clique size at most k?
- Given a graph G and a proper k-coloring c of G, does there exist a supergraph G' of G that is properly colored by c and is a unit interval graph?

We show that those problems are polynomial for fixed k. On the other hand, we prove that the first problem is equivalent to deciding if the bandwidth of G is at most $k-1$. Hence, it is NP-hard and $W[t]$-hard for all t. We also show that the second problem is $W[1]$-hard. This implies that for fixed k, both of the problems are unlikely to have an $O(n^\alpha)$ algorithm, where α is a constant independent of k.

A central tool in our study is a new graph-theoretic parameter closely related to pathwidth. An unexpected useful consequence is the equivalence of this parameter to the bandwidth of the graph.

Key words. design and analysis of algorithms, parameterized complexity, interval graphs, bandwidth, pathwidth

AMS subject classifications. 68Q25, 68R10, 05C78, 05C85, 03D15, 68Q15

1. Introduction.

This paper studies the following graph-theoretic questions.

- *Problem* A. Given a graph G and a constant k, does there exist a supergraph G' of G that is a unit interval graph and has clique size at most k?
- *Problem* C. Given a graph G and a proper k-coloring c of G, does there exist a supergraph G' of G that is properly colored by c and is a unit interval graph?

A related problem that generalizes both of these problems is the following.

- *Problem* B. Given two graphs G and G^2 such that $G \subseteq G^2$, is there a unit interval graph G' such that $G \subseteq G' \subseteq G^2$ and G' has clique size at most k?

Those questions arise as abstractions of practical problems in molecular biology, as will be explained later. They are NP-complete, but we show that they are all polynomial when the parameter k is fixed. We prove that Problem A is equivalent to the BANDWIDTH problem. In particular, this equivalence together with a recent result of Bodlaender, Fellows, and Hallet [4] imply that Problems A and B are hard for the parameterized complexity class $W[t]$ for all t. We prove here that Problem C is $W[1]$-hard. This implies that none of the problems is likely to have an algorithm with time complexity bounded by $f(k)n^\alpha$ for any constant α.

Problem B may be viewed as a *sandwich problem*, since G' must be "sandwiched" between G and G^2. Sandwich problems were introduced in [22] and studied further in [20]. Problem B was shown to be NP-hard for the case where $k = |V|$ (i.e., without any restriction on the clique size) in [21]. The NP-hardness results here are stronger since they apply to Problem C, a restriction of Problem B in which the forbidden edges are those between like-colored endpoints.

* Received by the editors November 5, 1993; accepted for publication (in revised form) September 26, 1994. Portions of this paper were presented at the 34th Annual IEEE Symposium on the Foundations of Computer Science, Santa Fe, NM, 1994.

† Department of Computer Science, Princeton University, Princeton, NJ 08544 (hkl@cs.princeton.edu).

‡ Department of Computer Science, Sackler Faculty of Exact Sciences, Tel Aviv University, Tel-Aviv 69978, Israel (shamir@math.tau.ac.il). The research of this author supported in part by a grant from the Ministry of Science and Technology, Israel.

Let A', B', and C' be analogous to Problems A, B, and C, respectively, with the only change that the supergraph G' is required to be interval instead of unit interval. Problem A' is equivalent to asking if the *pathwidth* of G is at most $k - 1$, and it arises in various guises (and under different names) in numerous areas (cf. [33]). It is NP-hard [2, 24, 28, 35] but linearly solvable for fixed k [3, 30]. Problem C' (and hence also B') was shown to be NP-hard when $k = |V|$ independently in [21] and [13]. Fellows, Hallet, and Wareham [13] also showed that the Problem C' is hard for $W[1]$ and that it is not finite state for bounded pathwidth or treewidth and, hence, not polynomially solvable by conventional algorithmic techniques. Recently, Bodlaender, Fellows, and Hallet [4] strengthened these results by proving that Problem C' is hard for $W[t]$ for all t. Note that the known results on Problems A' and C' do not directly imply analogous results for Problems A and C or vice versa. For example, as we show here, the parametric complexities of Problems A' and A are qualitatively very different.

The three problems that we study here arise in molecular biology. In order to study a genome, several copies of it are cut or broken down, and some of the resulting shorter segments (called *clones*) are preserved for further analysis. Depending on the technique used, the preserved clones may have variable length, or they may all have essentially the same length. In the process of producing the clones, all information about their relative position along the DNA chain is lost. The goal of *physical mapping* of DNA is to reconstruct that order, based on experimental data on the overlaps between pairs of clones [8, 26, 36]. Hence, the graph with vertices corresponding to clones and edges corresponding to overlapping pairs of clones should be an interval graph if clone lengths vary or a proper interval graph if all clones have the same length. Both types of graphs can be recognized in linear time [7, 10, 32].

In practice, information on overlap is never perfect. There are several ways to model this imperfection: If one assumes that all the errors in the data are false nonoverlaps (*false negatives*) and wishes to minimize their number, one gets the *interval graph completion problem* [17, Prob. GT35; 27], that is, minimize the number of edges whose addition to the graph will form an interval graph. If one assumes that all errors are erroneous overlaps (*false positives*), the *interval graph deletion problem* comes up [18]. If some pairs of clones are definite overlaps, some are definite nonoverlaps, and the rest are unknown, then one gets the *interval sandwich problem* [22]. All three problems are NP-hard, for interval and proper interval graphs. Hence, the question arises if introduction of additional biological constraints can make any of the problems tractable.

An important feature of real biological data is that the "width" of the map is consistently very small; the largest number of mutually overlapping clones is typically between 5 and 15, compared with a total number of clones in the thousands [31, 37]. Our study here focuses on the tractability of the models above in this special situation. More formally, do the problems remain NP-hard when the clique size of the proper interval graph is assumed to be bounded by a small constant? Hence, the model in Problem A bounds the width of the map and assumes false negatives only. Problem B allows a finer partition of the overlap information (into sure, unsure, and forbidden overlaps) while again bounding the map width. In Problem C, the set of clones consists of k disjoint subsets, with each subset originating from the dissection of a single copy of the genome. This implies that within each subset all clones are disjoint. Equivalently, the vertices corresponding to such a subset may all be assigned the same color. Admittedly, these models are crude, but they provide a starting point by allowing rigorous analysis. We believe that this line of research may eventually lead also to practical, tractable models, especially in view of the positive results given here for the fixed parameter cases.

The rest of the paper is organized as follows. In §2 we give formal definitions and background. In §3 we give two characterizations of proper pathwidth, a new graph-theoretic parameter introduced here, and in particular prove that proper pathwidth equals bandwidth. This result may be of independent interest, and we expect that it will be useful elsewhere. It implies immediately the NP-completeness of Problem A, its polynomiality for fixed k, and its $W[t]$-hardness for all t. Section 4 expands the equivalence to sandwich instances. Section 5 then exploits this equivalence to obtain a polynomial algorithm for Problem B (and hence C) when k is fixed. Section 6 contains a proof that the parameterized version of Problem C is hard for $W[1]$. Section 7 contains some concluding remarks.

2. Preliminaries.

In this section we give definitions and background and state without proof some well-known facts about notions that we will need in the sequel. For other graph-theoretical definitions see, e.g., [19].

2.1. Basic definitions.

All graphs are assumed to be undirected, simple, and finite. A graph $G = (V, E)$ is a *supergraph* of the graph $G' = (V', E')$ if $V = V'$ and $E \supseteq E'$. For a subset $W \subseteq V$, the subgraph of G *induced by* W is (W, E_W), where $E_W = \{(u, v) \in E \mid u, v \in W\}$. A *clique* in a graph $G = (V, E)$ is a set $C \subseteq V$ such that the subgraph induced by C is complete. A clique is *maximal* if it is not properly contained in any other clique in the graph. The *clique size* of G, denoted $\omega(G)$, is the maximum cardinality of a clique in G. An *independent set* in G is a set of vertices no two of which are adjacent. A *k-coloring* of graph $G = (V, E)$ is a function $c : V \to \{1, \ldots, k\}$ such that $c(u) \neq c(v)$ if $(u, v) \in E$. A supergraph G of G' *respects* the k-coloring c of G' if c is also a k-coloring for G.

2.2. Interval and proper interval graphs.

Let $S = (U, <)$ be a linear order. For every $u, w \in U$ such that $u \leq w$, the set $[u, w] = \{v \in U \mid u \leq v \leq w\}$ is called an *interval*. The most commonly used example involves intervals on the real line, but we shall refer also to intervals on finite, linearly ordered sets. A graph $G = (V, E)$ is an *interval graph* if one can associate with each vertex v an interval $I(v)$ such that two intervals intersect if and only if their vertices are adjacent. The set of intervals $\Sigma = \{I(v)\}_{v \in V}$ is called an (interval) *representation* for G, and G is called the *intersection graph* of Σ. A graph is a *proper interval graph* if it has an interval representation in which no interval is properly contained in another. A graph is a *unit interval graph* if it is an interval graph that has a representation on the real line in which all the intervals have equal length. Roberts has shown that these two definitions coincide as follows.

THEOREM 2.1. (See [38].) *A graph is unit interval if and only if it is proper interval.* □

The proof of the following lemma is straightforward and hence omitted.

LEMMA 2.2. *Every interval graph (and every proper interval graph) has an interval (resp., proper interval) representation on the real line in which all endpoints are distinct, and the left endpoints are assigned to the integers* $1, 2, \ldots, |V|$. □

We shall call such a representation *canonical.*

2.3. Pathwidth and proper pathwidth.

A *path decomposition* of a given graph $G = (V, E)$, is a sequence of subsets of V, $X = (X_1, \ldots, X_l)$ such that the following hold:

(1) $V = \cup_i X_i$.

(2) For each edge $(u, v) \in E$, there exists some $i \in \{1, \ldots, l\}$ so that both u and v belong to X_i.

(3) For each $v \in V$ there exist some $s(v), e(v) \in \{1, \ldots, l\}$ so that $s(v) \leq e(v)$, and $v \in X_j$ if and only if $j \in \{s(v), s(v)+1, \ldots, e(v)\}$.

The *width* of X is defined by $pw_X(G) = \max\{|X_i| \mid i = 1, \ldots, l\} - 1$. The *pathwidth* of G, denoted $pw(G)$, is the minimum value of $pw_X(G)$ over all path decompositions; i.e., $pw(G) = \min\{pw_X(G) \mid X$ is a path decomposition of $G\}$. The notion of pathwidth was originally introduced by Robertson and Seymour in [39].

LEMMA 2.3. (See, e.g., [6].) *For any path decomposition $X = (X_1, \ldots, X_l)$ of a graph G, every clique in G must be contained in some X_i.* □

The PATHWIDTH problem is to decide for a given graph G and a given integer k if $pw(G) \leq k$. Equivalent problems arise in various areas, including VLSI layout, processor management, node searching, and vertex separation (see the excellent survey of Möhring on pathwidth and the relations among these problems [33]). PATHWIDTH is NP-complete on arbitrary graphs [2, 28] and even for chordal graphs [24] and (using the equivalence to node search [29]) for planar graphs with vertex degrees at most three [35]. On the other hand, it is polynomial when k is fixed. The results of Robertson and Seymour [39] imply nonconstructively that an $O(n^2)$ algorithm exists for fixed k [14, 15]. The recent results of Bodlaender [3] yield a linear algorithm for PATHWIDTH for every fixed k [30, Chap. 11]. The notions of pathwidth and interval graphs are related by the following well-known observation.

LEMMA 2.4. (Cf. [33].) *For every graph G, the pathwidth of G is one less than the least clique size of any interval supergraph of G.* □

We introduce here the notion of a *proper path decomposition* of G; it is as a path decomposition X that satisfies (1)–(3) and also

(4) For every $u, v \in V$, $\{s(u), s(u)+1, \ldots, e(u)\} \not\subset \{s(v), s(v)+1, \ldots, e(v)\}$.

(As usual, $\subset$ denotes strict containment.) The *proper pathwidth* of G, denoted $ppw(G)$, is the minimum value of $pw_X(G)$ over all proper path decompositions of G; namely, $ppw(G) = \min\{pw_X(G) \mid X$ is a proper path decomposition of $G\}$.

Clearly, $pw(G) \leq ppw(G)$, but note that $pw(G)$ and $ppw(G)$ can differ dramatically: for the star graph with $2n+1$ vertices, $G = K_{1,2n}$, $pw(G) = 1$ but $ppw(G) = n$. An easy way to see the last equality is to the characterizations we shall provide in the next section.

2.4. Bandwidth. Another measure of graph width that we shall use is bandwidth; that is, for $G = (V, E)$ with $|V| = n$, a linear *layout* of the graph is a 1–1 function $L : V \rightarrow \{1, \ldots, n\}$. The *bandwidth of* L is $bw_L(G) = \max_{(u,v) \in E}\{|L(u) - L(v)|\}$. The *bandwidth* of G is the minimum bandwidth over all layouts of G; namely, $bw(G) = \min\{bw_L(G) \mid L$ is a layout of $G\}$. The BANDWIDTH problem is to decide for a given graph G and integer k, if $bw(G) \leq k$. This problem has been studied intensely because of its application to sparse matrix algebra [9]. It is known to be NP-complete even for binary trees [16] and for caterpillars with hair length at most three [34]. On the other hand, it is solvable in $O(n^k)$ for arbitrary k [23] and in linear time for $k = 2$ [16].

2.5. Sandwich problems. Given two graphs $G^1 = (V, E^1)$ and $G^2 = (V, E^2)$ such that G^2 is a supergraph of G^1, a graph $G = (V, E)$ is called a *sandwich* for this pair if $E^1 \subseteq E \subseteq E^2$. ($G$ must be "sandwiched" between G^1 and G^2, hence the name.) Let E^3 be the set of all edges in the complete graph with vertex set V that are not in E^2. We assume that a *sandwich instance* S is actually represented by the triplet (V, E^1, E^3). For a family $\mathcal{F}$ of graphs, the $\mathcal{F}$-SANDWICH problem is to decide for a given instance if there exists a sandwich graph $G \in \mathcal{F}$. For example, in the *proper interval sandwich* problem the set $\mathcal{F}$ is the set of all proper interval graphs.

Sandwich problems model flexibility (or ambiguity) of the problem data, where the edges of $E^2 - E^1$ are not known for sure to be included or excluded from the graph. Sandwich problems were introduced in [22] and studied later in [5, 13, 20, 21]. Their applications include matrix algebra, evolutionary tree construction, and temporal reasoning. Physical mapping of DNA motivated our study here of the following problem.

PROPER INTERVAL SANDWICH WITH BOUNDED CLIQUE SIZE (BPIS):
INPUT: A sandwich instance $S = (V, E^1, E^3)$ and an integer k.
QUESTION: Does there exist a sandwich G for S that is proper interval graph with $\omega(G) \le k$?

The proper interval sandwich problem (with no restriction on the clique size) was shown to be NP-complete in [21]. BPIS includes it as a special case by setting $k = |V|$. Hence, BPIS is NP-complete. In §5 we give an $O(f(k) \times |V|^{k-1})$ algorithm for BPIS, where $f(k)$ is a term that varies only with k. Thus, fixing k, one can determine if there exists a proper interval sandwich G with $\omega(G) \le k$ in polynomial time.

We shall also study the following special case of BPIS.

COLORED PROPER INTERVAL GRAPH COMPLETION (CPIC):
INPUT: A graph $G = (V, E)$ and a k-coloring c of G.
QUESTION: Does there exist a proper interval supergraph of G that respects c?

Note that CPIC is a restriction of BPIS: Given a graph $G = (V, E)$ with a k-coloring c, build the sandwich instance $S = (V, E, E^3)$ where $E^3 = \{(u, v) \mid u, v \in V, c(u) = c(v)\}$. Clearly, every supergraph of G which respects the coloring c is a sandwich graph for S that has clique size at most k, and vice versa.

In order to the describe this algorithm we need to extend the definition of graph bandwidth to sandwich instances as follows: Let $S = (V, E^1, E^3)$ be a sandwich instance. As before, a *layout* L of S is a one-to-one function from V onto $\{1, \ldots, n\}$. The *bandwidth of* L is $bw_L(S) = \max_{(u,v)\in E^1}\{|L(u) - L(v)|\}$. A layout L of a sandwich instance S is a *legal layout* if for every $(u, v) \in E^3$ there is no $(x, y) \in E^1$ such that $L(x) \le L(u) < L(v) \le L(y)$. The *bandwidth* of S is the minimum bandwidth over all legal layouts of G. Namely, $bw(S) = \min\{bw_L(S) \mid L \text{ is a legal layout of } S\}$. If S has no legal layout define $bw(S) = \infty$.

Remark 2.5. When $E^3 = \emptyset$, every layout of S is legal and $bw(S) = bw(G^1)$.

2.6. Parameterized complexity. Recently, Downey and Fellows initiated a systematic analysis of the complexity of parameterized decision problems [1, 11, 12]. An instance of a *parameterized decision problem* Π is a pair (x, k) where k is the parameter and x is the input other than the parameter, with $|x| = n$. The interest is usually in parameterized problems that are NP-complete but have polynomial complexity when the parameter k is fixed. This is the situation for CLIQUE, PATHWIDTH, BANDWIDTH, and many other problems. However, the dependence of the complexity on k may vary drastically. For some problems (like PATHWIDTH), algorithms of complexity $f(k)n^{\alpha}$ are known (for some constant α), while for others (like CLIQUE) the best known algorithms require $f(k)n^{g(k)}$ steps. Hence, for fixed k the complexity of the former is polynomial, and in some cases linear, independent of k, while in the latter the degree of the polynomial depends on k. A parameterized problem is called *fixed parameter tractable* if it belongs to the former family; i.e., it is solvable in time complexity $O(f(k)poly(n))$, where f is an arbitrary function of

k only and *poly* is a polynomial in n only, independent of k. The class of all fixed parameter tractable parameterized decision problems is denoted FPT. For example, PATHWIDTH and VERTEX COVER are NP-hard for variable k, but both are in FPT.

Downey and Fellows defined an appropriate notion of reduction for parameterized problems as follows: Let Π_1 and Π_2 be two parameterized decision problems. A *parameterized reduction* from Π_1 to Π_2 is an algorithm that, given an instance (x, k) for Π_1, decides if the answer is "Yes" in $O(f(k)poly(n))$ time, using an oracle for Π_2 on instances with parameter value no greater than $g(k)$, where *poly* is a polynomial in n only and f and g are arbitrary functions of k only. Thus, if Π_1 parametrically reduces to Π_2 and $\Pi_2 \in$ FPT, so is Π_1.

Parameterized decision problems are classified with respect to the W-hierarchy. To introduce it we need some definitions. A *mixed boolean decision circuit* is a boolean circuit with a single output and two types of gates: (1) *small gates:* AND, OR gates with fan-in two and NOT gates; and (2) *large gates:* AND and OR gates with unrestricted fan-in. The *weft* of a circuit C is the maximum number of large gates on an input–output path in C. The *depth* of C is the maximum number of gates (small or large) on an input-output path in C. Let F be a family of mixed boolean decision circuits. The yes-instances of the *parameterized problem associated with* F, denoted by Π_F, are $\{(C, k) \mid C \in F$ accepts an input vector with k ones$\}$. The class $W[t]$ consists of those parameterized decision problems that parametrically reduce to Π_F, where F is a family of mixed boolean decision circuits of bounded depth and weft at most t. This family of classes of parameterized decision problems satisfies FPT $\subseteq W[1] \subseteq W[2] \subseteq \cdots$ and is called the *W-hierarchy.* All containments are conjectured to be proper [11].

A problem Π is *$W[s]$-hard* if every problem in $W[s]$ has a parameterized reduction to Π. If Π is both in $W[s]$ and *$W[s]$-hard*, then Π is *$W[s]$-complete.* Thus, each problem that is hard for $W[s]$ for some $s \geq 1$ is conjectured not to have an algorithm with complexity bound $f(k)n^\alpha$. For example, DOMINATING SET is $W[2]$-complete [11], and BANDWIDTH was recently shown to be $W[t]$-hard for all t [4]. We shall use the latter result in §3. INDEPENDENT SET was shown to be $W[1]$-complete [1]. In §6 we give a parameterized complexity reduction from INDEPENDENT SET to CPIC. Hence, if CPIC has an $f(k)n^\alpha$ algorithm, then so does INDEPENDENT SET, and every other problem in $W[1]$, which is considered unlikely.

3. Characterizations of proper pathwidth.

In this section we show two equivalent characterizations of proper pathwidth. The following observation is analogous to Lemma 2.4.

LEMMA 3.1. *For every graph G, the proper pathwidth of G is one less than the least clique size of any proper interval supergraph of G.*

Proof. Suppose $X = (X_1, \ldots, X_l)$ is a proper path decomposition of G and $pw_X(G) = ppw(G) = k$. On the linear order $1 < 2 < \cdots < l$, let $I(v)$ be the interval $[s(v), e(v)]$. Then $\{I(v)\}_{v \in V}$ is a representation of an interval graph G'. G' is proper since no interval is strictly contained in the other, by property (4) in the definition of a proper path decomposition. G' is a supergraph of G, by property (2). Since X is also a path decomposition of G', Lemma 2.3 implies that every clique in G' must be contained in some X_i. Hence, $\omega(G') \leq k + 1$.

Conversely, suppose $k+1$ is the least clique size of any proper interval supergraph of G. Let $\{I(v)\}_{v \in V}$ be a proper interval representation of such a supergraph $G' = (V, E')$ with $\omega(G') = k+1$, in which all endpoints are distinct, and $I(v) = [l(v), r(v)]$. Order the $2n$ endpoints from left to right as $p_1, \ldots, p_{2n}$. Define $X_i = \{v \in V \mid p_i \in$

$I(v)\}$ for $i = 1, \ldots, 2n$. (Here and throughout $n = |V|$.) We claim that $(X_1, \ldots, X_{2n})$ is a proper path decomposition of G. Checking the four requirements from a proper path decomposition one can observe that (1) is immediate. (2) follows since $E \subseteq E'$, and if $(u, v) \in E'$ and $l(u) < l(v)$, then $l(v) \in I(v) \cap I(u)$. In other words, every edge in E' is represented in some X_i. Since G' is a proper interval graph, it follows that $s(v) = l(v)$ and $e(v) = r(v)$ satisfy (3) and (4). Since the clique size of G' is $k + 1$, each point p_i can meet at most $k + 1$ intervals in the representation. Hence, $\max_i |X_i| \leq k + 1$, so $ppw(G) \leq k$. □

We turn now to the proof of the main theorem in this section. It is actually a special case of a theorem about sandwich instances that will be proved in the next section (see Remark 4.3). However, since this theorem is of independent interest and its proof is simpler and more intuitive, we include its proof below.

THEOREM 3.2. *The bandwidth of a graph equals its proper pathwidth.*

Proof. For the graph $G = (V, E)$ with $ppw(G) = k$, by Lemma 3.1 there exists a supergraph $G' = (V, E')$ of G that is proper interval with clique size $k + 1$. Let $\{I(v)\}_{v \in V}$ be a canonical representation for G', where $I(v) = [l(v), r(v)]$. Define a layout L on G by $L(u) = l(u)$ for all $u \in V$. Suppose $L(u) - L(v) > k$ for some $(u, v) \in E$, where $L(v) = i$. Then $\{L^{-1}(i), \ldots, L^{-1}(i + k + 1)\}$ form a clique of size $k + 2$ in G', a contradiction. Hence, $bw(G) \leq bw_L(G) \leq ppw(G)$.

Conversely, let L be a layout of G so that $bw_L(G) = bw(G) = k$. Form a set of equal-length intervals on the real line $\{I(v)\}_{v \in V}$ where $I(v) = [L(v), L(v) + k]$. Let $G' = (V, E')$ be the intersection graph of that set of intervals. Clearly $E' \supseteq E$ since if $(u, v) \in E$, then $|L(u) - L(v)| \leq k$, so $I(u) \cap I(v) \neq \emptyset$. Since the maximum clique size for G' is at most $k + 1$, using Lemma 3.1, $ppw(G) \leq k$. Hence, $bw(G) \geq ppw(G)$. □

This interesting—and somewhat surprising—equivalence between bandwidth and proper pathwidth allows us to draw several immediate conclusions from the results on bandwidth.

COROLLARY 3.3.

A. *Finding the proper pathwidth is NP-hard, even for binary trees and for caterpillars with hair length at most three.*

B. *Problem* A *can be solved in $O(f(k)n^{k-1})$ time and, in particular, is polynomial when k is fixed.*

C. *Deciding whether the proper pathwidth of a graph is not greater than two can be done in linear time.*

D. *The parametric version of Problem* A *is $W[t]$-hard for every $t > 0$.*

E. *Problem* B *is NP-complete and its parametric version is $W[t]$-hard for every $t > 0$.* □

These results follow immediately from known results on bandwidth, using Theorem 3.2. Result A follows from [16, 34], B follows from [23], C from [16], and D from [4]. Result E follows since Problem B is a generalization of Problem A and thus at least as hard.

4. The bandwidth of a sandwich instance. We saw that one can embed a graph in a proper interval graph G' with $\omega(G') \leq k$ if and only if $bw(G) \leq k$. The question that we consider now is what happens when we cannot use every edge when forming the embedding and we have forbidden edges. In other words, we ask for a characterization of those sandwich instances that have a proper interval sandwich G' with $\omega(G') \leq k$. To state this characterization, we already generalized the notion of bandwidth to sandwich instances in §2 and we use it in the following theorem.

THEOREM 4.1. *Let $S = (V, E^1, E^3)$ be a sandwich instance. There is a proper interval sandwich $\tilde{G}$ in S with $\omega(\tilde{G}) \leq k+1$ if and only if $bw(S) \leq k$.*

Proof. Suppose $bw(S) \leq k$, and let L be a legal layout of S for which $bw_L(S) \leq k$. Let $|V| = n$ and $\epsilon = \frac{1}{n}$. For every vertex $v \in V$ in increasing order of $L(v)$ define recursively an interval $I(v) = [l(v), r(v)]$ as follows.

1. If $L(v) = 1$ define $I(v) = [1, r(v)]$ where $r(v) = \max(\{L(w) \mid (v,w) \in E^1\}) + \epsilon$.
2. Suppose $L(v) = p$, and $I(v)$ has already been defined for every vertex v with $L(v) < p$. Define $I(v) = [p, r(v)]$ where $r(v) = \max(\{L(z) \mid (v,z) \in E^1\} \cup \{r(L^{-1}(p-1))\}) + \epsilon$.

In particular, $L(v) = l(v)$ for every $v \in V$. One can easily prove by induction on p that all the endpoints of the intervals are distinct, and for every two vertices $v, w \in V$, if $L(v) < L(w)$, then $l(v) < l(w)$ and $r(v) < r(w)$. Thus, the intersection graph $\tilde{G} = (V, \tilde{E})$ defined by these n intervals is a proper interval graph. The following observation can easily be proved by induction on the left endpoints of the intervals.

OBSERVATION 4.2. *Let $I(v) = [l(v), r(v)]$ and let $w = L^{-1}(\lfloor r(v) \rfloor)$. There is a vertex u with $L(u) \leq L(v)$ such that $(u, w) \in E^1$.* □

We now prove that $\tilde{G}$ is a sandwich graph for S: By the construction of the intervals, $E^1 \subseteq \tilde{E}$. Suppose $E^3 \cap \tilde{E} \neq \emptyset$. Let (x, y) be an edge in $E^3 \cap \tilde{E}$. Without loss of generality assume that $l(x) < l(y)$ (so $L(x) < L(y)$). Since $(x, y) \in \tilde{E}$, it follows that $r(x) > l(y)$. Let $w = L^{-1}(\lfloor r(x) \rfloor)$. According to Observation 4.2 there is an edge $(u, w) \in E^1$, and u satisfies that $L(u) \leq L(x)$. This contradicts the assumption that L is legal since $(x, y) \in E^3$, $(u, w) \in E^1$, and $L(u) \leq L(x) < L(y) \leq L(w)$.

Suppose that $\tilde{G}$ has a clique C with more than $k+1$ vertices. Let x and y be the vertices in C with minimum and maximum values, respectively, under the layout L. Since $|C| > k+1$, $L(y) - L(x) > k$. $(x, y) \in \tilde{G}$ implies that $r(x) > L(y) = l(y)$, so according to Observation 4.2 there is an edge $(u, w) \in E^1$ where $L(u) \leq L(x)$ and $L(w) = \lfloor r(x) \rfloor \geq L(y)$, and that is a contradiction to the assumption that $bw_L(S) \leq k$.

To prove the converse, assume that there is a proper interval sandwich $\tilde{G} = (V, \tilde{E})$ for S with $\omega(\tilde{G}) \leq k+1$. Let $\{I(v)\}_{v \in V}$ be a canonical proper interval representation of $\tilde{G}$ where $I(v) = [l(v), r(v)]$. Define a layout L of S by $L(v) = l(v)$ for all $v \in$V. We claim first that L is a legal layout. Suppose on the contrary that there is an edge $(x, y) \in E^3$ and an edge $(u, v) \in E^1$ such that $L(u) \leq L(x) < L(y) \leq L(v)$. $\tilde{G}$ is a sandwich graph for S, so (u, v) must belong to $\tilde{E}$ and thus $l(v) < r(u)$. Since $\tilde{G}$ is proper interval, the left endpoints must appear in the same order as the right endpoints in the realization, namely $r(u) \leq r(x) < r(y) \leq r(v)$. Hence, all four intervals intersect at $l(v)$ and thus $\{u, x, y, v\}$ induce a clique in $\tilde{G}$. In particular, $(x, y) \in \tilde{E}$, a contradiction.

Finally, we show that $bw_L(S) \leq k$. Suppose, on the contrary that $bw_L(S) > k$. Let (u, v) be an E^1-edge such that $L(u) - L(v) > k$. Let $C = \{x \mid L(v) \leq L(x) \leq L(u)\}$. Note that $|C| > k+1$. $\tilde{G}$ is a sandwich graph for S, so $(u, v) \in \tilde{E}$. Arguments similar to the ones in the previous paragraph imply that C induces a clique in $\tilde{G}$, contradicting the assumption that $\omega(\tilde{G}) \leq k+1$. □

Remark 4.3. One can obtain Theorem 3.2 as a corollary to the theorem above by applying Theorem 4.1 to the sandwich instance $S = (V, E, \emptyset)$. In that case, every supergraph of G is a sandwich graph for S, so Theorem 3.2 follows by Remark 2.5.

5. A polynomial algorithm for fixed *k*. In this section we show that deciding if a given sandwich instance S admits a proper interval sandwich graph with clique size at most k can be done in $O(f(k)n^{k-1})$ steps. Hence, Problems B and C are polynomial when k is fixed. Our algorithm is based on the equivalence established in the previous sections with the bandwidth problem. For fixed k, Saxe [40] gave an $O(n^{k+1})$ algorithm that determines if the bandwidth of an n-vertex graph is at most k. Gurari and Sudborough [23] reduced its complexity to $O(n^k)$. We generalize that algorithm to the problem of deciding if the bandwidth of a sandwich instance is at most k. We assume without loss of generality that in the input S each vertex is incident on at most $2k$ E^1-edges and that $G^1 = (V, E^1)$ is connected.

We need the following definitions. Let $S = (V, E^1, E^3)$ be a sandwich instance. A *partial layout* of S is a 1–1 mapping from some nonempty $V' \subseteq V$ to $\{1, \ldots, |V'|\}$. V' is called the *domain* of L and denoted $D(L)$. In this section we shall use the term *layout* for a partial layout and *complete layout* for a layout L with $D(L) = V$. The *bandwidth* of L is $bw_L(S) = \max\{|L(u) - L(v)| \mid (u,v) \in E^1, u, v \in D(L)\}$. If $e = (x,y) \in E^1$, $x \in D(L)$ and $y \notin D(L)$, then e is called a *dangling edge* in L. The set of all dangling edges in L will be denoted $d(L)$. A vertex in the domain of L is called *active* if it is incident on a dangling edge. Let $r = \min(\{i \mid L^{-1}(i) \text{ is active}\})$. The sequence $(L^{-1}(r), L^{-1}(r+1), \ldots, L^{-1}(|D(L)|))$ is called the *active region* of L and denoted $R(L)$. Note that the dangling edges of L are exactly the edges in the cut $(D(L), V - D(L))$ in G^1. The connectivity assumption guarantees that $d(L) = \emptyset$ if and only if the layout is complete.

A layout L is called *legal* if the following requirements hold.

1. For each E^3-edge with $u, v \in D(L)$ there is no $(x,y) \in E^1$ such that $x, y \in D(L)$ and $L(x) \le L(u) < L(v) \le L(y)$.

2. No E^3-edge has both endpoints in $R(L)$.

A layout L' *augments* the layout L if $D(L) \subseteq D(L')$ and $L'(u) = L(u)$ for all $u \in D(L)$. L can be augmented to a complete legal layout L' with $bw_{L'}(S) \le k$ only if (1) L is legal, (2) $bw_L(S) \le k$, and (3) $|R(L)| \le k$. We thus restrict the algorithm to construct only layouts in Λ_k where

$$\Lambda_k = \{L \mid L \text{ is a legal layout such that } bw_L(S) \le k \text{ and } |R(L)| \le k\}.$$

For $L_1, L_2 \in \Lambda_k$, define $L_1 \approx L_2$ if $R(L_1) = R(L_2)$ and $d(L_1) = d(L_2)$. $\approx$ is an equivalence relation that partitions Λ_k into *classes*. For such a class C with $L \in C$, $R(C) = R(L)$ and $d(C) = d(L)$ are uniquely defined. Moreover, the assumption that G^1 is connected implies that equivalent layouts have identical domains, so $D(C)$ is also well defined for every class C. We call the pair $\Sigma(C) = (R(C), d(C))$ the *signature* of the class C, and we call a signature *valid* if a class with that signature exists. Given a sandwich instance S and integer k, $bw(S) \le k$ if and only if there is a class C in Λ_k such that $\Sigma(C) = (\emptyset, \emptyset)$. This class contains all the legal layouts in Λ_k.

5.1. The algorithm. The algorithm is based on the dynamic programming technique. It generates systematically all classes in Λ_k, until the search is exhausted or a class corresponding to a complete legal layout with bandwidth at most k is found. It is based on the following observation.

OBSERVATION 5.1. *Let $X = (x_1, \ldots, x_l)$, $l \le k$, and let $\hat{d} \subseteq E^1$ be a set of edges, each incident on one vertex from X. There is a class C' in Λ_k with $\Sigma(C') = (X, \hat{d})$ and $|D(C')| > 1$ if and only if for every $1 \le i \le l-1$, $(x_i, x_l) \notin E^3$ and there exists a class C in Λ_k such that $R(C) = (z_1, \ldots, z_j, x_1, \ldots, x_{l-1}), 0 \le j \le k - l + 1$, and*

$$d(C) = \hat{d} + \{(x_l, y) \in E^1 \mid y \in R(C)\} - \{(x_l, y) \in E^1 \mid (x_l, y) \in \hat{d}\}.$$

Proof. Suppose a class C' with $|D(C')| > 1$ exists, and let $L' \in C'$. Since L' is legal, for every $1 \le i \le l-1$, $(x_i, x_l) \notin E^3$. Since C' is in Λ_k, $bw_{L'}(S) \le k$. Clearly $L'(x_l) = |D(L')|$. Let L be a layout with $D(L) = D(L') - \{x_l\}$ such that L' augments

L. It is easy to see that L is legal and $bw_L(S) \leq k$. Thus, the class to which L belongs will be the required class C. The proof of the converse is similar. □

Under the conditions of the observation we shall say that C' *extends* C. Hence, the observation says that a nonsingleton class exists if and only if it extends another class.

Let $V = \{v_1, \ldots, v_n\}$. The algorithm starts by initializing a queue Q to contain n classes $C_1, \ldots, C_n$ where $R(C_i) = \{v_i\}$ and $d(C_i)$ contains all the E^1-edges incident on v_i. In each iteration a class is removed from Q and every class in Λ_k that extends it and was not treated before is generated and added to Q. This process ends in one of two possible ways.

1. If $bw(S) \leq k$, then the class C whose active region is empty (and then, by the connectivity of G^1, $D(C) = V$) is generated in some stage of the algorithm. When this happens the algorithm stops with a positive answer.
2. If $bw(S) > k$, then the algorithm will stop with a negative answer when its queue becomes empty.

A prototype of the algorithm is depicted in Figure 1.

```
ALGORITHM BANDWIDTH($S, k$);

/* $S = (V, E^1, E^3)$ is a sandwich instance */
/* initialization */
for each $v \in V$ :
        1. Create an equivalence class $C$
        with $R(C) = \{v\}$ and $d(C) = \{(v, y) \mid (v, y) \in E^1\}$.
        2. Mark $C$ and add it to the queue $Q$.
begin
        while $Q \neq \emptyset$ do :
        /* iteration */
        begin
                    3. Remove $C$ from $Q$.
                    4. Let $T$ be the set of classes in $\Lambda_k$ that extend $C$.
                    5. for each $C' \in T$ do :
                         5.1 if $R(C') = \emptyset$ output "$bw(S) \leq k$" and stop.
                         5.2 if $C'$ is unmarked mark $C'$ and add it to $Q$.
        end
        4. Output "$bw(S) > k$" and stop.
end
```

FIG. 1. *An algorithm for determining if a sandwich instance has bandwidth at most k.*

Let us explain how the set of all classes that extend C (denoted T in step 4 of the algorithm of Figure 1) can be generated. Distinguish the following two cases.

(A) If $|R(C)| = k$ there is at most one class C' that extends C. If $R(C) = (x_1, \ldots, x_k)$, such C' can exist only if exactly one dangling edge, say (x_1, y), is incident on x_1. In that case, C' must have $R(C') = (x_i, \ldots, x_k, y)$, where x_i is the first vertex in $R(C)$ that is connected by a dangling edge to a vertex other than y, and $d(C') = d(C) - \{(x_j, y) \mid (x_j, y) \in d(C)\} + \{(y, z) \in E^1 \mid z \notin D(C)\}$. Class C' exists if and only if y is not connected by an E^3-edge to any vertex in the set $\{x_2, \ldots, x_k\}$.

(B) If $|R(C)| = l < k$, up to $|V - D(C)|$ classes may extend C: Let $R(C) = (x_1, \ldots, x_l)$ and let $y \in V - D(C)$. Suppose x_i, $1 \leq i \leq l$, is the first vertex in $R(C)$ that is connected to a vertex other than y by a dangling edge. Then, if y is not connected by an E^3-edge to any vertex in $\{x_j | 1 \leq j \leq l\}$, the class C' with $R(C') =$

$(x_i, \ldots, x_l, y)$ and $d(C') = d(C) - \{(x_j, y) \mid (x_j, y) \in d(C)\} + \{(y, z) \mid z \notin D(C)\}$ exists and extends C.

5.2. Implementation. We describe an implementation of the algorithm that runs in time and space complexity $O(f(k)n^k)$. A class C in Λ_k will be represented by its signature $\Sigma(C)$. We assume that the E^3 edges are recorded in a vertex–vertex incidence matrix, so that the existence of an E^3 edge between two given vertices can be checked in constant time. Constructing an incidence matrix from a representation of E^3 by adjacency lists takes $O(n^2)$ time and space, so for $k \geq 2$ such a stage would not increase the worst case bounds of the algorithm. (The case $k = 1$ is trivial since $bw(S) = 1$ if and only if the graph G^1 is a path.)

LEMMA 5.2. *Given a class C and a vertex $y \notin D(C)$, one can verify whether there exists a class C' with $D(C') = D(C) \cup \{y\}$ in constant time assuming k is fixed. If C' exists, its signature can also be constructed in constant time.*

Proof. Since $|R(C)| \leq k$, verifying that y is not connected by an E^3 edge to any vertex in the active region takes constant time. To compute $\Sigma(C')$ given $\Sigma(C)$, one must copy each vertex from the region of C starting from the first vertex that is connected by a dangling edge to a vertex other than y. For each such vertex, E^1-edges that connect it to y should be removed from the new list of dangling edges. Since $|d(C)|$ is bounded from above by a function of k, this stage takes constant time. Vertex y should be added as the last vertex in the region. The set of new dangling edges, $\{(y, z) \mid z \notin D(C)\}$, can be constructed in constant time even though $D(C)$ is not represented explicitly since $\{(y, z) \mid z \notin D(C)\} = \{(y, z) \mid z \notin R(C)\}$. This construction takes constant time since the number of E^1-edges incident on y is at most $2k$. □

Let us now analyze the overall complexity of the algorithm for fixed k. Recall that in G^1 all the degrees are bounded by $2k$. Hence, for every active region R the number of possible sets of dangling edges with active region R is bounded by a constant. Therefore, the number of classes with active region of size k is $O(n^k)$, and the number of classes whose active region is smaller than k is $O(n^{k-1})$.

The algorithm inserts into the queue only classes that are new, i.e., previously unmarked. In order to do that without an increase in the time complexity one can maintain an array of all possible class candidates and their status as marked or unmarked, using $O(n^k)$ space. The marks enable the algorithm to encounter a class only once.

For a class with an active region of size k, there is only one possible class that extends it and by Lemma 5.2 a constant time is needed to check whether this class exists and if so to generate it. For a class with a smaller active region, there are up to n possible extensions. All the vertices in $V - D(C)$ are candidates for extension of C, but $D(C)$ is not maintained explicitly. To find that set, note that after deleting all dangling edges from $G^1 = (V, E^1)$, one obtains a disconnected graph $\tilde{G}^1$ in which $D(C)$ and $V - D(C)$ must be disconnected. The vertices in $V - D(C)$ constitute exactly these connected components of $\tilde{G}^1$ that do not contain an active vertex. Enumerating all candidates for extension can be done by depth first search of the appropriate components in $\tilde{G}^1$ in $O(n)$ steps [41], since $|E^1| = O(n)$. Checking each candidate and adding it to Q if it actually exists can be done in a constant time by Lemma 5.2. Thus, we obtain that the overall time complexity of the algorithm is $O(n^k)$ using no more than $O(n^k)$ space. The following theorem summarizes our findings.

THEOREM 5.3. *Given a sandwich instance S the algorithm determines whether $bw(S) \leq k$ in $O(f(k)n^k)$ time and space.* □

Together with Theorem 4.1 we obtain the following corollary.

COROLLARY 5.4. *Problem* B (*and hence also Problem* C) *can be solved in* $O(f(k)n^{k-1})$ *time and space and, in particular, in polynomial time and space when k is fixed.* □

By maintaining a representative layout with every class generated by the algorithm, one can obtain a complete legal layout with bandwidth not greater than k when the algorithm halts with a positive answer. The cost is an increase by a factor of n in the space complexity of the algorithm. The proof of the "if" part of Theorem 4.1 demonstrates how to obtain from that layout a unit interval sandwich graph in $O(n)$ time.

6. Colored proper interval graph completion is $W[1]$-hard. In this section we prove the following theorem.

THEOREM 6.1. *The Colored Proper Interval Graph Completion Problem is hard for* $W[1]$.

To prove Theorem 6.1 we describe a parameterized reduction from INDEPENDENT SET. Given a graph $G = (V, F)$ and an integer k as inputs to INDEPENDENT SET, we build a graph $G' = (V', F')$ colored using only $f(k) = 8k + 7$ colors. The graph G' is built such that it has a unit interval supergraph that respects the same coloring if and only if G has a size k independent set. Assume that $V = \{1, \ldots, n\}$ and $F = \{e_1, \ldots, e_m\}$. For convenience, we assume that the edge $e_t = (i, j)$ is denoted such that $i < j$.

6.1. Constructing G'. Let us first give an overview of the construction (see Figure 2): It consists of a *platform*—a long chain in which certain vertices are replaced by cliques, with subblocks corresponding to edges, and of k *floating paths*—each one is a long chain corresponding to a vertex in the independent set. Each path also consists of subblocks corresponding to edges. The platform and the floating paths are all connected at their endpoints, which forces their intervals to overlap in the representation of any unit interval graph completion. An additional decision component called *arrow* is connected to each edge-subblock in the platform. It is designed so that it prevents the complete placement of the k floating paths on top of the platform if and only if the graph does not contain an independent set of size k.

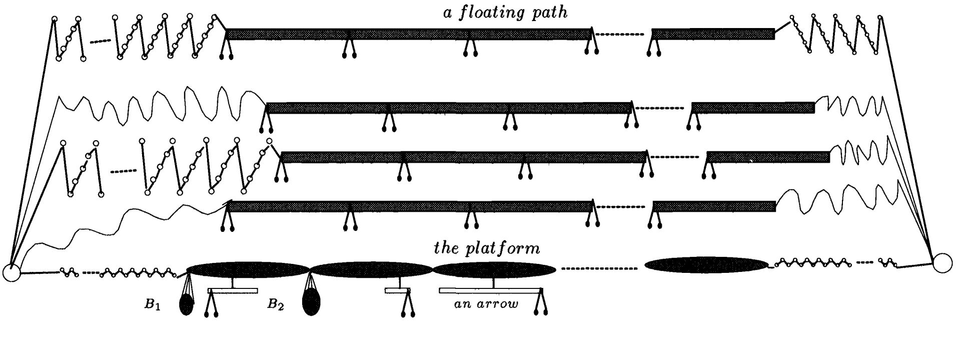


FIG. 2. *The graph G'.*

To describe these components we introduce the following definitions. Let $G^1 = (V^1, E^1)$ and $G^2 = (V^2, E^2)$ be two colored graphs. A colored graph G is a *concatenation* of G^1 and G^2 if it can be constructed by taking their union and identifying pairs of vertices, where in each pair one vertex is from V^1, the other from V^2, and their colors are identical. The concatenation is completely specified by these pairs of vertices. We shall occasionally omit listing the pairs in a concatenation when they are clear from the context. A graph G is a *concatenation* of $G_1, \dots, G_s$ if one can construct it by concatenating a graph H with G_s where H is a concatenation of $G_1, \dots, G_{s-1}$. If G is a concatenation of the colored graphs $H_1, \dots, H_s$, we denote by $H_j(G)$ the copy of H_j in G. If X is a set of vertices in a graph G, then $X(G)$ will denote the subgraph induced by X in G. For x a vertex in H and H' some copy of H, $x(H')$ will denote the vertex x in the copy H' of H. Greek letters will denote colors, small Latin letters will denote vertices, and capital Latin letters will usually denote graphs.

6.1.1. The floating paths. We first construct k isomorphic but differently colored floating paths. Each path consists of left and right ends and a middle part. Figure 3 gives an overview of the structure of such a path.

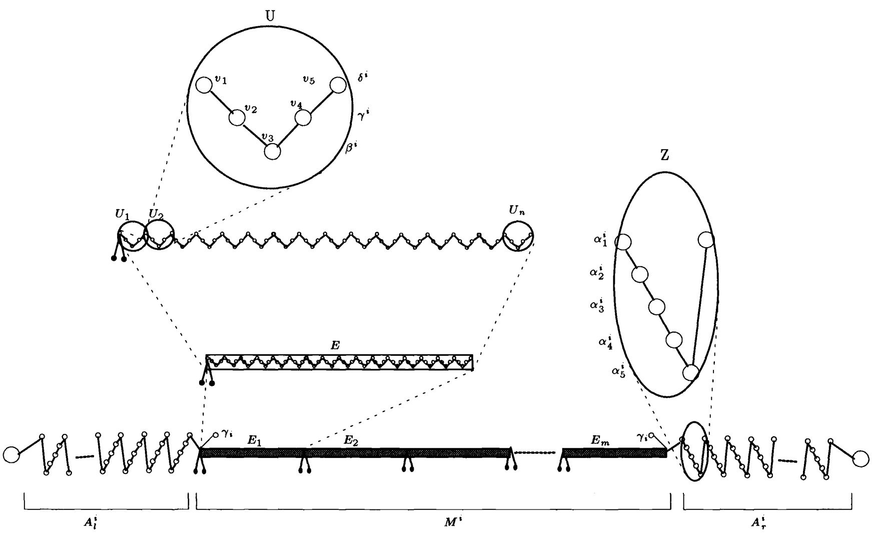


FIG. 3. *The structure of the floating path P^i.*

First we describe the construction of the middle part, which is denoted M. This part will overlap with the middle part of the platform in a representation of any unit interval completion. Let U denote a path with five vertices: $v_1, v_2, \dots, v_5$. Color it such that $c(v_1) = c(v_5) = \delta$, $c(v_3) = \beta$, $c(v_2) = c(v_4) = \gamma$. Take n copies of this colored U path denoted $U_1, U_2, \dots, U_n$ and *concatenate* them by identifying $v_5(U_j)$ with $v_1(U_{j+1})$ for every j, $1 \leq j \leq n-1$. Denote the resulting path E. Now, take m copies of E denoted $E_1, \dots, E_m$ and concatenate them by identifying $v_5(U_n(E_j))$ with $v_1(U_1(E_{j+1}))$ for $1 \leq j \leq m-1$. This resulting graph is the middle part M. Equivalently, one can think about M as a concatenation of nm copies of U or as a path of $4nm+1$ vertices colored periodically $\delta, \gamma, \beta, \gamma, \delta, \gamma, \dots$.

Next we describe the right and left ends of a floating path. These identical parts actually let the path "float" on the platform; i.e., using them we gain some flexibility in the starting position of M relative to the platform. Let Z denote a path on six vertices denoted $z_1, \dots, z_6$. Color it such that $c(z_1) = c(z_6) = \alpha_1$, $c(z_i) = \alpha_i, 2 \leq i \leq 5$.

Concatenate n copies of Z denoted $Z_1, \ldots, Z_n$ by identifying $z_6(Z_j)$ with $z_1(Z_{j+1})$ for every $1 \leq j \leq n-1$. Call the resulting graph A.

Complete a construction of a path P as follows. Take two copies of A, A_l, and A_r (we shall call these the *Left Accordion* and the *Right Accordion*, respectively) and one copy of M. Connect $z_1(Z_1(A_l))$ by an edge to $v_1(U_1(E_1))$ and connect $z_1(Z_1(A_r))$ by an edge to $v_5(U_n(E_m))$.

Each of the k floating paths denoted $P^1, \ldots, P^k$ is isomorphic to the path P above, where in P^i a private set of colors $\alpha_1^i, \alpha_2^i, \ldots, \alpha_5^i, \delta^i, \gamma^i, \beta^i$ replaces $\alpha_1, \alpha_2, \ldots, \alpha_5, \delta, \gamma, \beta$, respectively. Denote the subpaths of P^i as the corresponding subpaths of P with the additional superscript i (e.g., M^i, E_j^i).

On each P^i hang vertices colored ξ as follows: two such vertices are connected to $v_1(U_1(E_j^i))$ for every $1 \leq j \leq m$. We shall call them the *cherries* of E_j^i. Also, connect two vertices colored γ^i one to $v_1(U_1(E_1^i))$ and the other to $v_5(U_n(E_m^i))$.

6.1.2. The platform. Similar to a floating path, the platform also consists of three parts: a middle part similar to the M part of a floating path and two flanking end parts. Certain vertices in the paths are replaced by cliques in the platform, the end parts are shorter, and colored to allow little flexibility in any unit interval representation. The structure of the platform is depicted in Figure 4.

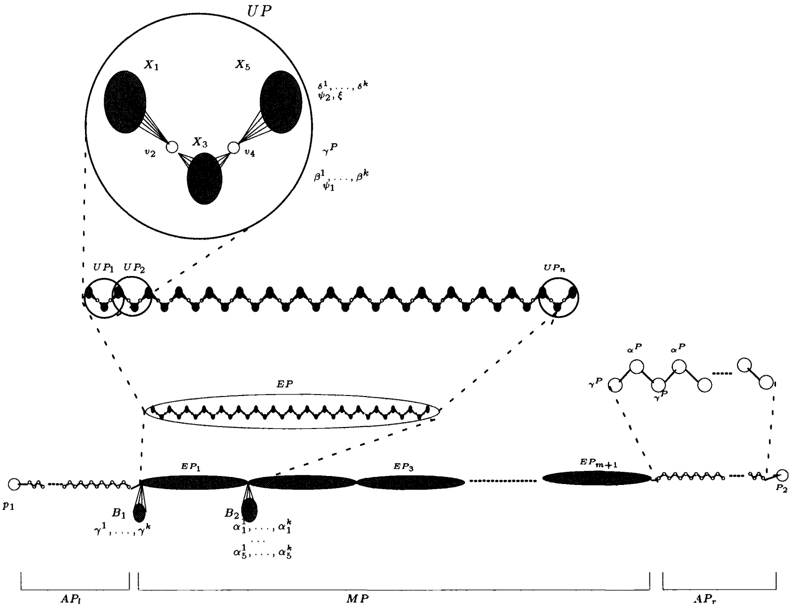


FIG. 4. *The structure of the platform.*

The main component of the platform is the graph UP that can be constructed from the path U as follows. Replace v_3 with a clique X_3 on $k+1$ vertices, and replace v_1 and v_5 with cliques X_1, X_5, respectively, each containing $k+2$ vertices. Connect each vertex of X_3 to v_2 and v_4, each vertex of X_1 to v_2, and each vertex of X_5 to v_4.

Color UP as follows: $c(v_2) = c(v_4) = \gamma^P$. X_3 is colored such that it contains one vertex colored β^i for every $1 \leq i \leq k$ and a vertex colored ψ_1. X_1 and X_5 are colored such that they contain one vertex colored δ^i for every $1 \leq i \leq k$, a vertex colored ψ_2, and a vertex colored ξ.

To construct the middle part of the platform, first take n copies of UP denoted $UP_1, \ldots, UP_n$ and concatenate them by identifying vertices colored identically in $X_5(UP_j)$ and $X_1(UP_{j+1})$ for every $1 \leq j \leq n-1$. Denote the resulting graph EP. Next, take $m+1$ copies of EP, denoted $EP_1, \ldots, EP_{m+1}$, and concatenate them by identifying like-colored vertices in $X_5(UP_n(EP_j))$ and $X_1(UP_1(EP_{j+1}))$ for every $1 \leq j \leq m$. Denote the resulting concatenation MP. One can also refer to MP as a concatenation of $n(m+1)$ copies of UP obtained by identifying vertices colored identically in $X_5(UP_j)$ and $X_1(UP_{j+1})$ for every $1 \leq j \leq n(m+1)-1$.

To define the flanking ends of the platform, let AP be a path with $n-1$ vertices $z_1^p, \ldots, z_{n-1}^p$, colored such that $c(z_i^p) = \gamma^p$ if i is odd and $c(z_i^p) = \alpha^p$ for i even. (As we shall see later, this coloring forces such a path to "spread out" with no additional overlap in any representation.) Take MP and two copies of AP, AP_l, and AP_r (which will be called the *Left end* and the *Right end* of the platform). Connect $z_1^p(AP_l)$ to all vertices of $X_1(UP_1(EP_1))$ and connect all vertices of $X_5(UP_n(EP_{m+1}))$ to $z_1^p(AP_r)$. The construction of the platform is now complete.

6.1.3. The arrows. An additional graph called an arrow that is "almost a path" is generated for each edge in the original graph G. Each arrow is connected to a certain clique in the subpath of the platform corresponding to its edge. The arrow (and, in particular, two special vertices at one of its ends) is designed to force the independence of the vertices that the t floating paths will define. Figure 5 demonstrates the details of this part in the construction.

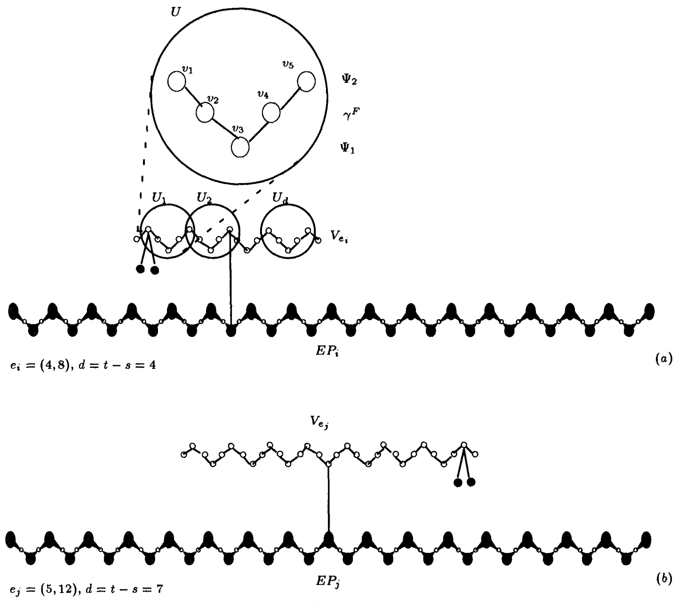


FIG. 5. *An arrow and its location on the platform:* (a) $d = t - s$ *is even,* $s \notin K$; (b) $d = t - s$ *is odd,* $s \in K$, $t \notin K$.

For each edge $e = (s,t) \in F$ construct an *arrow* V_e as follows. Set $d = t - s$; and let $U_1, \ldots, U_d$ be d copies of a path U colored such that $c(v_1) = c(v_5) = \psi_2$, $c(v_3) = \psi_1$, and $c(v_2) = c(v_4) = \gamma^F$. Concatenate them by identifying $v_5(U_j)$ with $v_1(U_{j+1})$ for every $1 \leq j \leq d-1$. Call $v_1(U_1(V_e))$ the *head* of the arrow and call $v_5(U_d(V_e))$ the *tail* of the arrow. Connect two vertices colored ξ to the head, call them the *cherries* of the arrow V_e. Connect one vertex colored γ^F to the head of the path and connect another such vertex to the tail. Let $m(V_e)$ be the middle vertex of the arrow, i.e., $m(V_e) = v_5(U_{\frac{d}{2}}(V_e))$ if d is even and $m(V_e) = v_3(U_{\frac{d+1}{2}}(V_e))$ if d is odd. Note that $c(m(V_e)) = \psi_2$ if d is even, and otherwise $c(m(V_e)) = \psi_1$.

For each edge e_j connect $m(V_{e_j})$ to all the vertices of $X_3(UP_{\frac{t+s}{2}}(EP_j))$ if d is even (note that there is no vertex colored ψ_2 in $X_3(UP_{\frac{t+s}{2}}(EP_j))$) and to all the vertices of $X_5(UP_{\frac{t+s-1}{2}}(EP_j))$ if d is odd (note that there is no vertex colored ψ_1 in $X_5(UP_{\frac{t+s-1}{2}}(EP_j))$.

6.1.4. Completing the construction. To complete the construction of G' take two more cliques B_1 and B_2: B_1 contains k vertices colored with the k colors $\gamma^i, 1 \leq i \leq k$; and B_2 contains $5k$ vertices colored α^i_j, $1 \leq j \leq 5$, $1 \leq i \leq k$. Connect every vertex of B_1 with every vertex of $X_1(UP_1(EP_1))$ and every vertex of B_2 with every vertex of $X_1(UP_1(EP_2))$. The role of these two cliques is to force the representation of each middle part in a floating path to start in between their representations in any unit interval graph completion.

Finally, to connect the k floating paths and the platform, introduce two vertices p_1, p_2 colored θ^p. Connect $z_6(Z_n(A^i_l))$, $1 \leq i \leq k$, and $z^p_{n-1}(AP_l)$ to p_1, and connect $z_6(Z_n(A^i_r))$, $1 \leq i \leq k$, and $z^p_{n-1}(AP_r)$ to p_2.

The overall structure of G' is drawn schematically in Figure 2.

6.2. Validity of the construction. The construction described is clearly polynomial. We will now prove that G' can be turned to a unit interval graph by adding edges that respect its coloring if and only if G has a size k independent set.

6.2.1. $\Leftarrow$ (if). First suppose that G has a size k independent set K. We shall show how to represent each vertex in G' by a closed unit interval on the real line such that the following two requirements hold.

1. Intervals that correspond to vertices with the same color do not overlap.
2. Intervals that correspond to adjacent vertices in G' do overlap.

The intersection graph defined by this representation of the intervals will be the required properly colored unit interval supergraph of G'.

For every vertex $x \in V'$, $I(x)$ will denote the interval that corresponds to it. If this unit interval is placed between c and $c+1$ on the real line denote it by $I(x) = [l(x), r(x)] = [c, c+1]$. For the two intervals I_1 and I_2, define that $I_1 < I_2$ if and only if $r(I_1) < l(I_2)$. For I an interval and b a point on the real line, define $I < b$ $(b < I)$ if and only if $r(I) < b$ $(b < l(I))$.

In every copy of a U path or UP graph in our construction $c(v_2) = c(v_4)$. Thus, in any representation that fulfills the requirements above $I(v_2) \cap I(v_4) = \emptyset$. A representation of a U path or a UP graph is called *right-oriented* if and only if $I(v_2) < I(v_4)$ and otherwise it is *left-oriented.* We shall occasionally say that "a graph is right-oriented" for short when we actually mean that its representation is right oriented.

Let Y be a UP graph on the platform and fix a right-oriented representation for Y. With respect to this representation define two points on the real line, $Right(Y)$ and $Left(Y)$, as $Right(Y) = \min(\cap_{x \in X_5(Y)} I(x))$ and $Left(Y) = \max(\cap_{x \in X_1(Y)} I(x))$. Note that since $X_5(Y)$ and $X_1(Y)$ are cliques, $\cap_{x \in X_5(Y)} I(x) \neq \emptyset$ and $\cap_{x \in X_1(Y)} I(x) \neq \emptyset$, and thus $Right(Y)$ and $Left(Y)$ are well defined. Say that a unit interval I *is in* Y and denote it $I \lhd Y$ if and only if $Left(Y) < I < Right(Y)$. For a vertex x in G', x *is in* Y if and only if $I(x) \lhd Y$. Denote it $x \lhd Y$.

6.2.1.1. Placing the platform. The whole representation will occupy the interval [0, 4nm + 6n]. First assign $I(p_1) \mapsto [0, 1]$. For every vertex v on the platform whose distance (length of shortest path) from p_1 is d assign $I(v) = [d, d+1]$. Thus, the intervals of the vertices in the Left end are assigned to the right of the interval $I(p_1)$; then, further to the right, the intervals of the vertices in MP are placed; and finally, the intervals corresponding to the vertices in the Right end of the platform are placed. In particular, $I(p_2) = [4nm + 6n - 1, 4nm + 6n]$. Interval $I(p_2)$ will be the rightmost interval of the representation. Note that all the intervals corresponding to the vertices in one of the cliques X_1,X_3, or X_5 in any of the UP graphs on the platform are assigned to the same position. Also note that the representation of all the UP graphs on the platform is right-oriented.

Remark 6.2. Let Y be a UP graph on the platform. Obviously $Right(Y) - Left(Y) = 3$. Moreover, for every vertex $x \in X_5(Y) \cup \{v_4(Y)\}$, $Right(Y) \in I(x)$ and for every vertex $x \in X_1(Y) \cup \{v_2(Y)\}$, $Left(Y) \in I(x)$. Since there is no vertex colored ξ in X_3 but there is one in X_1 and X_5, the maximum number of other vertices colored ξ that can be placed in each UP graph on the platform is two.

6.2.1.2. Placing the cliques B_1 and B_2. Assign the vertices of B_1 to the same position as the vertices of $X_1(UP_1(EP_1))$. The vertices of B_2 are assigned to the same position as the vertices of $X_1(UP_1(EP_2))$.

6.2.1.3. Placing the arrows. Let $e_j = (s,t)$ be an edge in G, $d = t - s$, and let V_{e_j} be the corresponding arrow. Assign $m(V_{e_j})$ to the same position as that of the clique on the platform to which it is connected.

Since K is an independent set, it cannot contain both s and t. If $s \notin K$ place the arrow such that the representation of each U path on the arrow is right-oriented and $v_1(U_i(V_{e_j})) \lhd UP_{s+i-1}(EP_j)$ for every $1 \leq i \leq d$. The two cherries of V_{e_j} that are connected to its head are placed in $UP_s(EP_j)$ (see Figure 5(a)). If $s \in K$ then place the arrow such that the representation of each U path is left-oriented and $v_1(U_i(V_{e_j})) \lhd UP_{t-i+1}(EP_j)$ for every $1 \leq i \leq d$. In the latter case the two cherries are placed in $UP_t(EP_j)$ (see Figure 5(b)).

One should note that the cherries of the arrow of (s,t) are placed in $UP_s(EP_j)$ if $s \notin K$ and in $UP_t(EP_j)$ otherwise. Thus, $UP_i(EP_j)$ does not contain vertices colored ξ originating from an arrow, for every $1 \leq j \leq m$ and $i \in K$.

6.2.1.4. Placing the floating paths. Let $K = \{i_1, \ldots, i_k\}$. The j-th floating path, P^j, will correspond to i_j. First M^j is placed and then the left and right accordions A^j_l and A^j_r are placed as follows.

Each U path in P^j is placed such that its representation is right-oriented and $v_1(U_l(M^j)) \lhd UP_{i_j+l-1}(MP)$ for $1 \leq l \leq n \times m$. Thus, the whole representation of M^j spreads from $UP_{i_j}(EP_1)$ to $UP_{i_j}(EP_{m+1})$ on the platform. Recall that to each $v_1(U_1(E^j_s))$, $1 \leq s \leq m$, two cherries are connected. These cherries are placed together with $v_1(U_1(E^j_s))$ in $UP_{i_j}(EP_s)$.

This placement can be carried out without violating requirement 1 above since in each of the UP graphs, X_3 does not contain a vertex colored γ^j and X_1, X_5 do not contain vertices colored β^j. One also must check that in each UP graph no more than two vertices colored ξ are placed. This is indeed the situation since for every $1 \leq j \leq m$ the arrows insert vertices colored ξ only to $UP_l(EP_j)$ for indices $l \notin K$. On the other hand, the floating paths insert vertices colored ξ to $UP_l(EP_j)$ for every $l \in K$ and $1 \leq j \leq m$. Since all the other vertices in the arrows have colors that do not appear in P^j, property 1 is satisfied.

The left accordion A^j_l should be placed between $I(p_1)$ and $I(v_1(U_1(M^j)))$. According to the placement of the paths, $n+1 < l(I(v_1(U_1(M^j)))) < 5n-1$. Analyzing the coloring of A^j_l one can verify that for each number x such that $n+1 < x < 5n-1$

a representation of A_l^j that spreads out on an interval whose length is x can be constructed. Such a representation can be placed between $I(p_1)$ and $I(v_1(U_1(M^j)))$ since there is no other vertex colored α_i^j, $1 \leq i \leq 5$, placed there. A representation of the right accordion can similarly be constructed and placed between $I(v_5(U_{n\times m}(M^j)))$ and $I(p_2)$.

The intervals that correspond to the two vertices colored γ^j connected to $v_1(U_1(E_1^j))$ and $v_5(U_1(E_m^j))$ are placed with $I(v_1(U_1(E_1^j)))$ and $I(v_5(U_1(E_m^j)))$ in $UP_{i_j}(EP_1)$ and $UP_{i_j}(EP_{m+1})$, respectively.

6.2.2. ⇒ (only if). Suppose G' has a unit interval supergraph G'' that respects the coloring of G'. Our goal now is to prove that under these conditions there exists an independent set of size k in G. We actually show that the structure and coloring of G' force every representation of G'' to look very similar to the one we built in the previous part of the proof.

Let $\{I(x)\}_{x\in V'}$ be a representation of G'' by closed unit intervals. Since $c(p_1) = c(p_2) = \theta^p$, $I(p_1) \cap I(p_2) = \emptyset$. One can assume without loss of generality that $I(p_1) < I(p_2)$ (otherwise just reverse the entire representation).

The platform contains paths from p_1 to p_2 in which every second vertex is colored γ^P. Every such path starts with AP_l, ends with AP_r, and in between contains all the vertices colored γ^P in MP, a representative from $X_1(UP_i(MP))$ and $X_3(UP_i(MP))$, for each i, $1 \leq i \leq n(m+1)$, and a representative from $X_5(UP_{n(m+1)}(MP))$. Since every second vertex on such a path is colored γ^P, its representation cannot "fold back" and all the intervals of the vertices colored γ^P must be disjoint and appear (from left to right) in the same order as they appear in the path. This immediately implies that the representation of each UP graph on the platform is right-oriented.

Let X and Y be two right-oriented UP graphs or two identically colored right-oriented U paths. We say that X is *to the left of* Y or, equivalently, Y is *to the right of* X if and only if $I(v_4(X)) < I(v_2(Y))$. Denote this situation by $X \prec Y$ or $Y \succ X$. (We use a different notation here since the sets of intervals corresponding to X and Y may overlap.) The existence of a path between p_1 and p_2 on which every second vertex is colored γ^P implies that there must exist a linear order between the UP graphs in the platform, i.e., $UP_j(MP) \prec UP_{j+1}(MP)$ for every $1 \leq j \leq n(m+1)-1$.

Let $b_1 = \max(\cap_{x\in B_1 \cup X_1(UP_1(EP_1))} I(x))$ and $b_2 = \min(\cap_{x\in B_2 \cup X_1(UP_1(EP_2))} I(x))$. Note that the assumption $I(p_1) < I(p_2)$ implies $I(p_1) < b_1 < b_2 < I(p_2)$.

LEMMA 6.3. *$b_1 < I(v_1(U_1(M^i))) < b_2$ for every $1 \leq i \leq k$.*

Proof. First we prove that $b_1 < I(v_1(U_1(M^i)))$ for every $1 \leq i \leq k$. Fix i. Clearly $b_1 \notin I(v_1(U_1(M^i)))$, since $X_1(UP_1(EP_1))$ contains a vertex y colored δ^i, the same color of $v_1(U_1(M^i))$, such that $b_1 \in I(y)$. Suppose on the contrary that $I(v_1(U_1(M^i))) < b_1$. Recall that B_1 contains a vertex z colored γ^i such that $b_1 \in I(z)$. Since every second vertex on M^i is colored γ^i, the whole representation of M^i will spread out to the left of b_1. In particular, $I(v_5(U_{n\times m}(M^i))) < b_1$. This implies that the representation of the right accordion A_r^i should intersect b_2. That is a contradiction since it means that two intervals of vertices colored α_j^i for some $1 \leq j \leq 5$ overlap in b_2.

One can similarly show the other inequality for every i. First, note that $b_2 \notin I(v_1(U_1(M^i)))$ since $X_1(UP_1(EP_2))$ contains a vertex z with the same color as $v_1(U_1(M^i))$ such that $b_2 \in I(z)$. Assume on the contrary that $b_2 < I(v_1(U_1(M^i)))$. Thus, $I(p_1) < b_2 < I(v_1(U_1(M^i)))$ and the representation of the left accordion A_l^i intersects b_2, a contradiction. □

LEMMA 6.4.

(a) *$U_j(M^i)$ is right-oriented for every $1 \leq j \leq nm$ and $1 \leq i \leq k$.*

(b) *$U_j(M^i) \prec U_{j+1}(M^i)$ for every $1 \le j \le nm-1$ and $1 \le i \le k$.*

Proof. Fix i. Since in M^i every second vertex is colored γ^i, its representation cannot "fold back." Thus, either all the U paths on P^i are right-oriented and satisfy $U_j(M^i) \prec U_{j+1}(M^i)$, $1 \le j \le nm-1$, or they are all left-oriented and satisfy $U_{j+1}(M^i) \prec U_j(M^i)$, $1 \le j \le nm-1$.

Suppose the second possibility occurs. Together with Lemma 6.3 this implies that the whole representation of M^i is to the left of b_2. In particular, $I(v_5(U_{nm}(M^i))) < b_2$ which means that the representation of the right accordion A^i_r intersects b_2 and that is a contradiction. □

Remark 6.5. Each vertex in M^i (including the first and the last) that is not colored γ^i has two adjacent vertices that are colored γ^i. This forces the endpoints of the intervals in each U path in M^i to satisfy $l(I(v_1)) < l(I(v_2)) < l(I(v_3)) < l(I(v_4)) < l(I(v_5))$. Similarly the placement of the intervals corresponding to vertices in a UP graph on the platform must be such that $l(I(x_1)) < l(I(v_2)) < l(I(x_3)) < l(I(v_4)) < l(I(x_5))$ for every $x_1 \in X_1$, $x_3 \in X_3$, $x_5 \in X_5$.

So far we proved that the representation of each M^j starts, at its left end, in some $UP_{i_j}(EP_1)$ (i.e., $v_1(U_1(M^j)) \lhd UP_{i_j}(EP_1)$) for some $1 \le i_j \le n$ and spreads to the right. Recall that to each $v_1(U_1(E^j_s)$, $1 \le j \le k$, $1 \le s \le m$, two cherries colored ξ are connected. The intervals corresponding to them must be placed in the same UP graph as $v_1(U_1(E^j_s))$ itself, since there are vertices colored ξ in X_1 and X_5 of every UP graph.

Three vertices colored ξ cannot be placed in the same UP graph. Thus, for every $j_1 \ne j_2$, M^{j_1} and M^{j_2} must start in different UP graphs. We obtain that the starting positions of M^j, $1 \le j \le k$, in EP_1 on the platform define a set of k distinct vertices $K = \{i_1, \dots, i_k\}$ in G. We shall prove that K is an independent set.

In the following lemma we analyze further the structure of a floating path's representation and prove that it must be intertwined in the platform in a very specific manner.

LEMMA 6.6. *If $v_1(U_j(M^i)) \lhd UP_s(MP)$, then $v_1(U_{j+1}(M^i)) \lhd UP_{s+1}(MP)$ for every $1 \le j \le nm-1$ and $1 \le s \le n(m+1)-1$.*

Proof. Suppose that $v_1(U_j(M^i)) \lhd UP_s(MP)$. According to Lemma 6.4, $v_1(U_{j+1}(M^i))$ cannot be in $UP_l(MP)$ for some $l < s$. Suppose the claim is false. There are two possibilities for failure.

(1) $v_1(U_{j+1}(M^i)) \lhd UP_s(MP)$. This implies, according to Remark 6.5 and Lemma 6.4, that $v_3(U_j(M^i))$, which is colored β^i, must also be in $UP_s(MP)$. Let z be the vertex colored β^i in $X_3(UP_s(MP))$. Clearly $I(z) \cap I(v_3(U_j(M^i))) = \emptyset$. Assume that $I(z) < I(v_3(U_j(M^i))) < Right(UP_s(MP))$. Since $Right(UP_s(MP)) - r(z) \le |I(v_4(UP_s(MP)))| = 1$, this assumption leads to a contradiction. Assuming that $Left(UP_s(MP)) < I(v_3(U_j(M^i))) < I(z)$, since $l(z) - Left(UP_s(MP)) \le |I(v_2(U_j(M^i)))| = 1$, this also implies a contradiction. Thus $v_1(U_{j+1}(M^i))$ cannot be also in $UP_s(MP)$.

(2) $Right(UP_{s+1}(MP)) < I(v_1(U_{j+1}(M^i)))$. Let x and y be the vertices colored γ^i in $X_1(UP_{s+1}(MP))$ and $X_5(UP_{s+1}(MP))$, respectively. Since we assumed that $v_1(U_j(M^i)) \lhd UP_s(MP)$, the order of the intervals in the representation is $I(v_1(U_j(M^i))) < I(x) < I(y) < I(v_1(U_{j+1}(M^i)))$. Let z be the vertex colored β^i in $X_3(UP_{s+1}(MP))$. According to Remark 6.5 $l(x) < l(z) < r(z) < r(y)$, one gets that $z \lhd U_j(M^i)$. Consider now the interval $I(v_3(U_j(M^i)))$. Since $c(v_3(U_j(M^i))) = c(z)$, $I(z) \cap I(v_3(U_j(M^i))) = \emptyset$. The two possibilities $I(z) < I(v_3(U_j(M^i)))$ and $I(v_3(U_j(M^i))) < I(z)$ will both lead to a contradiction since $l(I(v_1(U_{j+1}(M^i)))) - r(I(v_3(U_{j+1}(M^i))))$ and $l(I(v_3(U_j(M^i)))) - r(I(v_1(U_j(M^i))))$ cannot be greater than

one. □

The immediate conclusion from this lemma and the discussion preceding it is that if M^j starts in $UP_{i_j}(EP_1)$, then $v_1(U_1(E_l^j)) \lhd UP_{i_j}(EP_l)$ for every $1 \leq l \leq m$. Thus, for every l, $1 \leq l \leq m$, $UP_{i_j}(EP_l)$ contains two cherries of P^j.

In order to prove that K is an independent set we should now check how the arrows are placed in our representation. Let V_{e_j} be an arrow corresponding to the edge $e_j = (s,t)$ and again let $d = t-s$. Recall that if d is even $v_5(U_{\frac{d}{2}}(V_{e_j})) = v_1(U_{\frac{d}{2}+1}(V_{e_j}))$ is connected to the vertices of $X_3(UP_{\frac{t+s}{2}}(EP_j))$. Thus, $v_1(U_{\frac{d}{2}+1}(V_{e_j})) \lhd UP_{\frac{t+s}{2}}(EP_j)$ and the representation of $U_{\frac{d}{2}+1}(V_{e_j})$ is either right-oriented or left-oriented. If d is odd, then $v_1(U_{\frac{d+1}{2}}(V_{e_j})) \lhd UP_{\frac{t+s-1}{2}}(EP_j)$ if the representation of $U_{\frac{d+1}{2}}(V_{e_j})$ is right-oriented, otherwise $v_1(U_{\frac{d+1}{2}}(V_{e_j})) \lhd UP_{\frac{t+s+1}{2}}(EP_j)$.

Analogously to Lemma 6.4 the following lemma holds with respect to the arrows.

LEMMA 6.7. *For every arrow V_e where $e = (s,t)$, all $U_j(V_e)$, $1 \leq j \leq t-s$, have the same orientation. Moreover, if the $U_j(V_e)$ are all right-oriented, then $U_j(V_e) \prec U_{j+1}(V_e)$ for every $1 \leq j < t-s$; otherwise, $U_{j+1}(V_e) \prec U_j^i(V_e)$ for every $1 \leq j < t-s$.* □

Let V_e be an arrow corresponding to the edge $e = (s,t)$. Define V_e to be *right-oriented* (*left-oriented*) if $U_j(V_e)$ is right-oriented (left-oriented) for every $1 \leq j \leq t-s$.

The arrows also must be intertwined in the platform, and the following lemma can be proved analogously to Lemma 6.6.

LEMMA 6.8. *Let V_{e_j} be an arrow corresponding to the edge $e_j = (s,t)$. If V_{e_j} is right-oriented, then $v_1(U_i(V_{e_j})) \lhd UP_{s+i-1}(EP_j)$ for every $1 \leq i \leq t-s$; and if V_{e_j} is left-oriented, then $v_1(U_i(V_{e_j})) \lhd UP_{t-i+1}(EP_j)$ for every $1 \leq i \leq t-s$.* □

Now suppose on the contrary that K is not an independent set, i.e., there is an edge $e_j = (i_s, i_t)$ where $i_s, i_t \in K$. In the discussion following Lemma 6.6 we already noted that each of $UP_{i_s}(EP_j)$ and $UP_{i_t}(EP_j)$ contains cherries originating in the floating paths P^s and P^t. But from Lemma 6.8 it follows that the two cherries connected to the arrow V_{e_j} must be placed together either in $UP_{i_s}(EP_j)$ or in $UP_{i_t}(EP_j)$. Since one cannot place more than two vertices colored ξ in any UP graph, we get a contradiction. Hence, K is an independent set of size k in G, and the proof is complete.

Since the parameterized reduction of Theorem 6.1 is actually a Karp reduction, we can also conclude with the following corollary.

COROLLARY 6.9. *The Colored Unit Interval Graph Completion Problem is NP-complete.* □

7. Concluding remarks.

In this paper we studied proper interval graph completion problems with small clique size. The problems arise in physical mapping of DNA. We showed that the Proper Interval Sandwich Problem with fixed bound k on the clique size is polynomial. On the other hand, we showed that a restriction of the problem, namely, the Colored Unit Interval Graph Completion Problem, with k viewed as a parameter is $W[1]$-hard and, hence, does not have an $O(f(k)n^\alpha)$ algorithm unless all problems in $W[1]$ (including, for example, INDEPENDENT SET) have one. This result also implies the NP-completeness of the problem.

Regarding the Proper Interval Graph Completion Problem With Minimum Clique Size, we showed that the problem is equivalent to BANDWIDTH and also to the new parameter *proper pathwidth*, which was defined here. This unexpected equivalence may be useful to other problems, since it allows one to apply tools from interval graph theory to bandwidth problems and vice versa. In fact, it has already proved its usefulness in our results here. It implied, using previous results on BANDWIDTH, that the problem when k is fixed is polynomial but $W[t]$-hard for all t in its parameterized version. This last hardness result is somewhat surprising since the analogous problem with interval graphs replacing proper interval graphs is known to be *linear* for fixed k and, in particular, fixed parameter tractable.

The algorithms presented here for fixed k are still impractical for the range of k required in the physical mapping problem. However, our results demonstrate that incorporating more biological restrictions into the model may cause the complexity to decrease. An interesting line of research is to introduce further realistic restrictions that will lead to efficient practical algorithms for physical mapping.

Acknowledgments. We thank Mike Fellows and Andrzej Proskurowski for their helpful comments on an early version of this manuscript. We also thank two anonymous referees for their careful remarks and suggestions.

REFERENCES

[1] K. Abrahamson, R. Downey, and M. Fellows, *Fixed-parameter intractability* II, in Proc. 10th Symposium on Theoretical Aspects of Computer Science, Lecture Notes in Comput. Sci. 665, Springer-Verlag, Berlin, 1993, pp. 374–385.

[2] S. Arnborg, D. J. Corneil, and A. Proskurowski, *Complexity of finding embedding in a k-tree*, SIAM J. Discrete Meth., 8 (1987), pp. 227–284.

[3] H. L. Bodlaender, *A linear time algorithm for finding tree-decomposition of small treewidth*, in Proc. 25th Symposium on the Theory of Computing, Association for Computing Machinery, New York, 1993, pp. 226–234.

[4] H. L. Bodlaender, M. R. Fellows, and M. T. Hallet, *Beyond NP-completeness for problems of bounded width: Hardness for the W hierarchy* (extended abstract), in Proc. 26th Symposium on the Theory of Computing, Association for Computing Machinery, New York, 1994, pp. 449–458.

[5] H. L. Bodlaender, M. R. Fellows, and T. J. Warnow, *Two strikes against perfect phylogeny*, in Proc. 19th International Comput. Algorithms Lang. Programming, W. Kuich, ed., Lecture Notes in Comput. Sci. 623, Springer-Verlag, Berlin, New York, Heidelberg, 1992, pp. 273–283.

[6] H. L. Bodlaender and R. H. Möhring, *The pathwidth and treewidth of cographs*, in Proc. 2nd Scandinavian Workshop on Algorithm Theory, 1990, Lecture Notes in Comput. Sci. 447, Springer-Verlag, Berlin, New York, Heidelberg, 1990, pp. 301–309.

[7] K. S. Booth and G. S. Lueker, *Testing for the consecutive ones property, interval graphs, and planarity using PQ-tree algorithms*, J. Comput. System Sci., 13 (1976), pp. 335–379.

[8] A. V. Carrano, *Establishing the order of human chromosome-specific DNA fragments*, in Biotechnology and the Human Genome, A. D. Woodhead and B. J. Barnhart, eds., Plenum Press, New York, 1988, pp. 37–50.

[9] E. Cuthill and J. McKee, *Reducing the bandwidth of sparse symmetric matrices*, in Proc. 24th National Conference of the Association for Computing Machinery, Association for Computing Machinery, New York, 1969, pp. 157–172.

[10] X. Deng, P. Hell, and J. Huang, *Linear-time representation algorithms for proper circular-arc graphs and proper interval graphs*, Technical Report, School of Computing Science, Simon Fraser University, Burnaby, BC, Canada, 1993; SIAM J. Comput., 25 (1996), pp. 390–403.

[11] R. G. Downey and M. R. Fellows, *Fixed-parameter intractability*, in Proc. Structures, IEEE Press, Los Alamitos, CA, 1992, pp. 36–49.

[12] ———, *Fixed-parameter tractability and completeness* III: *Some structural aspects of the W hierarchy*, in Complexity Theory: Current Research (Proc. 1992 Dagstuhl Workshop on Structural Complexity), K. Ambos-Spies, S. Homer, and U. Schöning, eds., Cambridge University Press, Cambridge, UK, 1993, pp. 191–226.

[13] M. R. Fellows, M. T. Hallet, and H. T. Wareham, *DNA physical mapping: Three ways difficult*, in Proc. European Symposium on Algorithms (ESA '93), Lecture Notes in Comput. Sci. 726, Springer-Verlag, Berlin, New York, Heidelberg, 1993, pp. 157–168.

[14] M. R. Fellows and M. A. Langston, *Nonconstructive advances in polynomial time complexity*, Inform. Process Lett., 26 (1987), pp. 157–162.

[15] ———, *On well-partial-order theory and its application to combinatorial problems of VLSI design*, SIAM J. Discrete Math., 5 (1992), pp. 117–126.

[16] 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.

[17] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman, San Francisco, 1979.

[18] P. W. Goldberg, M. C. Golumbic, H. Kaplan, and R. Shamir, *Four strikes against physical mapping of DNA*, J. Comput. Biol., 2 (1995), pp. 139–152.

[19] M. C. Golumbic, *Algorithmic Graph Theory and Perfect Graphs*, Academic Press, New York, 1980.

[20] M. C. Golumbic, H. Kaplan, and R. Shamir, *Graph sandwich problems*, J. Algorithms, 19 (1995), pp. 449–473.

[21] ———, *On the complexity of DNA physical mapping*, Adv. Appl. Math., 15 (1994), pp. 251–261.

[22] M. C. Golumbic and R. Shamir, *Complexity and algorithms for reasoning about time: A graph-theoretic approach*, J. Assoc. Comput. Mach., 40 (1993), pp. 1108–1133.

[23] E. Gurari and I. H. Sudborough, *Improved dynamic programming algorithms for the bandwidth minimization and the mincut linear arrangement problem*, J. Algorithms, 5 (1984), pp. 531–546.

[24] J. Gustedt, *On the pathwidth of chordal graphs*, Technical Report, Fachbereich Mathematik, Technische Universität Berlin, Berlin, Germany, 1992; Discrete Math., to appear.

[25] H. Kaplan, R. Shamir, and R. E. Tarjan, *Tractability of parameterized completion problems on chordal and interval graphs: Minimum fill-in and physical mapping* (extended abstract), in Proc. 35th Symposium on Foundations of Computer Science, IEEE Press, Los Alamitos, CA, 1994, pp. 780–791.

[26] R. M. Karp, *Mapping the genome: Some combinatorial problems arising in molecular biology*, in Proc. 25th Symposium of the Theory of Computing, Association for Computing Machinery, New York, 1993, pp. 278–285.

[27] T. Kashiwabara and T. Fujisawa, *An NP-complete problem on interval graphs*, in Proc. 12th IEEE Symposium of Circuits and Systems, IEEE Press, Piscataway, NJ, 1979, pp. 82–83.

[28] ———, *NP-completeness of the problem of finding a minimum-clique-number interval graph containing a given graph as a subgraph*, in Proc. 12th IEEE Symposium of Circuits and Systems, IEEE Press, Piscataway, NJ, 1979, pp. 657–660.

[29] L. M. Kirousis and C. H. Papadimitriou, *Searching and pebbling*, Theoret. Comput. Sci., 47 (1986), pp. 205–218.

[30] T. Kloks, *Treewidth*, Ph.D. thesis, Computer Science Department, Utrecht University, Utrecht, The Netherlands, 1993.

[31] Y. Kohara, K. Akiyama, and K. Isono, *The physical map of the whole E. coli chromosome: Application of a new strategy for rapid analysis and sorting of large genomic libraries,* Cell, 50 (1987), pp. 495–508.

[32] N. Korte and R. H. Möhring *An incremental linear time algorithm for recognizing interval graphs*, SIAM J. Comput., 18 (1989), pp. 68–81.

[33] R. H. Möhring, *Graph problems related to gate matrix layout and PLA folding,* in Computational Graph Theory, Computing Supplement 7, G. Tinhofer et al., eds., Springer, Vienna, 1990, pp. 17–51.

[34] B. Monien, *The bandwidth minimization problem for caterpillars with hair length* 3 *is NP-complete,* SIAM J. Algebraic Discrete Meth., 7 (1986), pp. 505–512.

[35] B. Monien and I. H. Sudborough, *Min cut is NP-complete for edge weighted trees*, Theoret. Comput. Sci., 58 (1988), pp. 209–229.

[36] R. Nagaraja, *Current approaches to long-range physical mapping of the human genome*, in Techniques for the Analysis of Complex Genomes, R. Anand, ed., Academic Press, London, 1992, pp. 1–18.

[37] M. V. Olson et al, *Random-clone strategy for genomic restriction mapping in yeast,* Proc. Nat. Acad. Sci. U.S.A., 83 (1986), pp. 7826–7830.

[38] F. S. Roberts, *Discrete Mathematical Models, with Applications to Social Biological and Environmental Problems.* Prentice–Hall, Englewood Cliffs, NJ, 1976.

[39] N. Robertson and P. D. Seymour, *Graph minors* I*: Excluding a forest*, J. Combin. Theory Ser. B, 35 (1983), pp. 39–61.

[40] J. B. Saxe, *Dynamic programming algorithms for recognizing small-bandwidth graphs in polynomial time*, SIAM J. Algebraic Discrete Meth., 1 (1980), pp. 363–369.

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

SIAM J. COMPUT.
Vol. 25, No. 3, pp. 562–599, June 1996

EFFICIENTLY PLANNING COMPLIANT MOTION IN THE PLANE*

J. FRIEDMAN[†], J. HERSHBERGER[‡], AND J. SNOEYINK[§]

Abstract. Any practical model of robotic motion must cope with the uncertainty and imprecision inherent in real robots. One important model is compliant motion, in which a robot that encounters an obstacle obliquely may slide along the obstacle. The authors start by investigating the geometry of compliant motion in the plane under perfect control and find a compact data structure encoding all paths to a goal. When the authors introduce uncertainty in control and position sensing, the same data structure allows them to find efficiently a compliant motion that reaches the goal, if one exists, to compute the boundary of the nondirectional backprojection of the goal, and to compute multistep plans for sensorless robots. This "preprocessing and query" approach has advantages of speed for online queries and flexibility for considering robots with different capabilities or initial positions in the same environment.

Key words. computational geometry, compliant motion, path hull data structure

AMS subject classifications. 68U05, 68P05

1. Introduction.

Planning the motion of a robot is a practical problem with many theoretically appealing variants [1, 4, 5, 13, 20, 26, 34]. Given a robot's initial and goal position in some environment, there are typically many paths connecting the initial position and the goal position. The desirability of a path depends not only on properties of the path (e.g., a shortest path [35] or a high-clearance path [31]) but also on the control and sensing abilities of the robot.

We investigate a motion paradigm called *compliant motion*, which is a mathematical idealization of several robot-control paradigms, including guarded move, generalized damper, and sliding mode control [1, Chap. 5]. A robot using compliant motion follows a "stable" path: even if the robot diverges slightly from the commanded path, it still reaches the goal. A point moving by compliant motion following a commanded direction α (with perfect control) travels through free space in direction α until it encounters an obstacle (see Figure 1). Then it slides along the obstacle until either friction is too large, or α no longer points into the obstacle. In the former case, it stops; in the latter, it resumes travel through free space in direction α. This type of motion enables the robot to "grope" its way towards the goal. In the basic model, the only way the robot can stop is by getting stuck [8, 9, 12]. In less restricted cases, some forms of goal sensing are possible [26].

When the robot's control is imperfect, the actual direction of the motion can deviate from α. We follow other researchers [8, 11, 26, 27] and assume that the deviation is bounded by some angle ϵ. For a given starting point, we look for all directions such that any motion whose instantaneous attempted direction deviates less than ϵ from the commanded direction is guaranteed to reach the goal.

Sliding on a boundary wall is "stable," since there is a range of directions that forces the robot to slide in the same way along that wall. Given a robot's control-

* Received by the editors February 15, 1994; accepted for publication (in revised form) October 6, 1994. This research was supported in part by Digital Equipment Corporation Systems Research Center. Preliminary versions of the material in this paper appeared in the 1989 ACM Symposium on Computational Geometry [15] and in the 1990 Scandinavian Workshop on Algorithm Theory [16].

[†] D. E. Shaw and Company, New York, NY 10036.

[‡] Mentor Graphics, 1001 Ridder Park Drive, San Jose, CA 95131.

[§] Department of Computer Science, University of British Columbia, Vancouver, BC V6T 1Z4, Canada. The research of this author was supported in part by an NSERC Research Grant and a fellowship from the British Columbia Advanced Systems Institute.

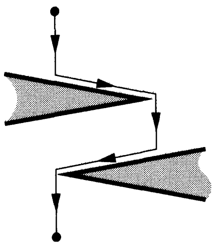


FIG. 1. *Compliant motion.*

uncertainty parameter ϵ, it may be possible to ensure that the robot always slides left (or right) along any particular wall that it encounters. Thus, it is sometimes possible to find a stable commanded direction that gets the robot all the way from the starting point to the goal. When there is no stable direction by which the robot can reach the goal, we may wish to outline a *plan,* which specifies a number of subgoals (regions of the environment), each one directly attainable from the previous one, that eventually leads to the goal. We would like to find a plan that contains as few subgoals as possible.

1.1. Previous work. In three-dimensional space, compliant motion planning is a provably difficult problem. It is hard for *PSPACE* [30] and for *NEXPTIME* [5]. Although work has been done on this problem [3], most results, including our own, deal with the more tractable case of compliant motion in the plane.

Lozano-Pérez, Mason, and Taylor [27] describe the *preimage backchaining* approach to compliant motion planning. Their idea is to compute the *preimage* or *backprojection* of the goal—the set of points in the environment from which the robot can reach the goal by following a single direction command. They treat the preimage as a new goal and compute its preimage. The second preimage contains all points that require a two-step plan to reach the original goal. They repeat this process until a region is found that contains the current position of the robot.

Erdmann [11] and Donald [8] describe algorithms for computing a *directional backprojection*: the set of points that reaches the goal by one motion in a given direction α with a given control uncertainly ϵ. Their algorithms run in $O(n \log n)$ time and $O(n)$ space for an environment composed of n line segments (of *size* n) and a goal of constant size. Latombe [26] surveys these algorithms and other issues and approaches.

When the direction α is not given, Donald [9] presents an $O(n^4 \log n)$ algorithm for the one-step case, and an $O(n^{r^{O(1)}})$ algorithm for the r-step case, in which each motion terminates with the robot getting stuck. Briggs [2] improves the single-step bound to $O(n^2 \log n)$.

1.2. Contributions of this paper. Our approach emphasizes two features: query formulations, and late introduction of control uncertainty.

Query formulations of motion planning problems are based on the assumption that the robot moves repeatedly in the same environment. We divide the problem into two phases: preprocessing of the environment, and queries of the form "In what directions can a robot at a point p reach the goal?" For an environment with k polygonal obstacles, whose total number of vertices is n, the preprocessing time is $O(kn \log n)$ and the query time is $O(k \log n)$. Even if used for a single query, the dependence on

kn (rather than on n^2) allows us to describe the obstacles in greater detail, using more points, without having to pay the quadratic cost of Briggs' method [2]. For multiple queries the savings is even greater.

We assume perfect control in the preprocessing phase and build data structures that efficiently encode the entire set of paths to the goal from every point in the polygon. The magnitude of the control uncertainty is part of our query data. This allows us to change the control uncertainty of the robot dynamically, or to consider multiple robots with different control capabilities and positions, without recomputing the data structures that support compliant motion calculations. For a starting point given at query time, we can also compute the maximum possible control uncertainty under which we can still guarantee that the robot reaches the goal. This does not increase the query time.

Our basic algorithm for the single-step planning problem has a number of applications and extensions. First, we can construct the boundary of the region from which a robot with imperfect control can reach the goal in a single step in $O(kn \log n)$ time. Second, we can answer queries that specify, instead of a starting point, a region guaranteed to contain the robot's starting position. We return the set of directions in which the robot can reach the goal no matter where in the region it starts. The query time is at most $O(kn)$, but is typically less. Third, we show how to extend our one-step results to the case of sensorless robots (robots that must stick at the goal) with no change to the algorithms, data structures, or running times. Finally, we consider two polynomial variants of the multistep planning problem: we solve a restricted version of the multistep planning problem for sensorless robots using $O(kn^2 \log n)$ preprocessing, $O(kn^2)$ space, and $O(kn \log n)$ query time, improving Donald's exponential bound [9] in this special case. When $k = 1$, we show how to solve the multistep problem for robots with perfect control and sensing with no change to the single-step complexity.

We employ a novel data structure called a *path hull*, which represents the convex hull of a simple polygonal chain (a *path*). Using the path hull data structure, one can find a tangent to the convex hull of the path, either from a point or parallel to a given line, in $O(\log n)$ time. Path hulls also support sequences of common maintenance operations at an amortized cost of $O(\log n)$ per operation. These sequences can consist of additions and deletions at the end of the path, and either path splits or joins (but not both splits and joins in the same sequence). We believe that path hulls are of independent interest.

The remainder of the paper is organized as follows. In the next section, we define our problem more precisely, and investigate the theoretical properties of compliant motion that we use in the algorithms. For clarity of the presentation, we describe the main algorithm in two steps. In §3 we present the algorithm for the special case in which the environment has no isolated obstacles (in other words, the environment is the interior of a simple polygon) and prove its correctness. In §4 we show how to extend the algorithm to the general case. In §5 we address the issues of imperfect control and position sensing, as well as the other applications of the basic algorithm. Section 6 describes in detail the path hull data structure used by the algorithms. We conclude in §7 with an open problem.

2. Fundamentals of compliant motion. In this section we define compliant motion and establish fundamental properties of paths followed by a robot doing compliant motion. We also investigate the α-backprojection, which is the set of points that reaches the goal when commanded to move in direction α.

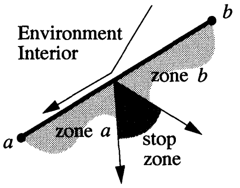


FIG. 2. *A friction cone.*

We first define notation. If a and b are points, we denote the direction from a to b by $\overrightarrow{ab}$. If α is a direction, then $\alpha^{\perp}$ is the direction 90° counterclockwise from α, and $-\alpha$ is the reverse direction. If α and β are directions, then the interval $(\alpha, \beta]$ denotes the range of directions from α counterclockwise to β. We later define a special direction 0 such that $\alpha < \beta$ means that $[\alpha, \beta]$ does not contain direction 0. If ϵ is an angle, then $\alpha + \epsilon$ is the direction ϵ counterclockwise from α. We use α^{+} to represent a direction that is infinitesimally counterclockwise from α, so that if $\alpha^{+} \leq \beta$ then $\alpha < \beta$. Similarly, α^{-} denotes a direction that is infinitesimally clockwise from α.

When the direction α is fixed within a given context, we generally choose the coordinate system so that α is parallel to the vector $(0, -1)$, and we say that α is pointing *down*. Within this frame of reference, the *up* direction is $-\alpha$, the *right* direction is $\alpha^{\perp}$, and the *left* direction is $-\alpha^{\perp}$.

2.1. Compliant motion trails. The environment, which we call P, is described by a set of k disjoint simple polygons that has a total of n vertices. Without loss of generality, we assume that one of the polygons, called the *outside polygon*, contains all the other polygons, called the *islands*. (We can always place a bounding polygon of constant complexity around an environment that has only islands, and ensure that the algorithm never uses the new walls for sliding.) The *interior* of the environment is the interior of the outside polygon excluding the islands. None of the islands needs contain any other island.

In this paper, the robot is a point. One can reduce more general robots to point robots by the *configuration space* approach of Lozano-Pérez and Wesley [28], in which a nonrotating convex robot corresponds to a reference point moving among configuration-space obstacles, and the configuration space can be computed in time linear in the size of the original environment (assuming the complexity of the robot itself is constant). The robot always moves through the interior of the environment, which we refer to as *free space*, or slides along a boundary edge of the environment (a *P-edge*).

Each P-edge has an associated *friction cone* that defines the robot's behavior when it hits the edge. Let $\overline{ab}$ be a P-edge, and assume that the interior of the environment is locally to the left of $\overrightarrow{ab}$. The friction cone of $\overline{ab}$ divides the direction interval $[\overrightarrow{ba}, \overrightarrow{ab}]$ into three zones: *zone a*, *zone b*, and the *stop zone*, which is a closed interval. (See Figure 2.) Suppose that a robot moving in direction α hits $\overline{ab}$. If α is in zone a, then the robot slides toward a; if α is in zone b, the robot slides toward b; and if α is in the stop zone, then the robot is stopped by friction when it hits $\overline{ab}$. Note that we do not require the friction cones to be symmetric, nor even require the stop zone to include the normal to $\overline{ab}$. When the stop zone does not contain the normal, the P-edge acts as a "conveyor belt."

When a robot sliding along a P-edge reaches an endpoint, it does one of three things: slides along an adjacent P-edge, leaves the boundary and goes into free space, or gets stuck. Suppose that $\overline{ab}$ and $\overline{bc}$ are two adjacent P-edges, and the robot has just finished sliding along $\overline{ab}$ toward b. If the commanded direction α falls in zone c of P-edge $\overline{bc}$, then the robot continues to slide. If α does not point into $\overline{bc}$, then the robot goes into free space at b, following direction α. Otherwise, the robot gets stuck at b, and we say that b is a *sticky vertex.* A vertex may also be sticky if the robot reaches it from free space: if the robot slides to a on $\overline{ab}$ and to c on $\overline{bc}$ then b is also considered sticky.

We introduce uncertainty in control in §5.1; before then, we assume perfect control. A starting point p and a commanded direction α define a unique path (a sequence of directed segments) that the robot follows, which we call a *trail* and denote by $T_\alpha(p)$. When there is no ambiguity, we may use T_α instead of $T_\alpha(p)$.

The preprocessing phase of our algorithm works with a fixed goal on the boundary of the environment. Throughout the rest of this paper, we assume that the goal is a vertex g; however, our results also hold for a goal that is a P-edge. When $T_\alpha(p)$ contains g, we say that "α is a *good* direction for p."

We conclude this section with two properties of compliant motion trails. The first is Observation 2.1, which is in the spirit of the "kinetic framework" of Guibas, Ramshaw, and Stolfi [19].

OBSERVATION 2.1. *Let the commanded direction α point vertically downward. If the leftmost (rightmost) segment l along T_α is not the first or last segment, then l is vertical and is directed down.*

Proof. If s and t are segments along T_α, and s has a positive horizontal component ("s is sloping to the right") while t has a negative horizontal component ("t is sloping to the left"), then there exists a free-space segment u along T_α between s and t (otherwise, there would be a sticky vertex in the middle of the trail). All free, space segments in T_α are directed down. □

LEMMA 2.2. *The trail T_α has no loops.*

Proof. Suppose that T_α has a loop. If T_α goes counterclockwise around the loop, then T_α must be directed up through the rightmost portion of the loop; if T_α goes clockwise around the loop, then T_α must be directed up through the leftmost portion of the loop. In either case, we get a contradiction with Observation 2.1. □

2.2. The α-backprojection.

Let α be a given direction. The *α-backprojection*, denoted by B_α, is the set of all points p for which α is a good direction. The α-backprojection plays an important role in our algorithms for compliant motion planning. In this section, we gain more insight into the structure of B_α.

Since α is fixed within the context of this section, we can assume that α points vertically downward. Thus we say that a point p is *above* point q when the vector $\overrightarrow{pq}$ has direction α. A P-edge $\overline{ab}$ of P is called *slidable towards a* if α is in zone a of $\overline{ab}$.

Figure 3 shows an example of an α-backprojection. In this figure, and in all the examples throughout this paper, the friction cones are the edge normals unless we explicitly state otherwise. Note that the boundary of B_α is composed of portions of P-edges and edges that extend through the free space of P. By the definition of sticky vertices (see §2.1 above), the free-space edges on the boundary of B_α are not part of B_α. (In the degenerate cases in which either of the P-edges incident to g is not slidable towards g, the free-space edge extending from g is part of B_α.) On the other hand, the P-edges on the boundary of B_α are part of B_α.

Note that the α-backprojection is "almost" a simple polygon. Although some of the free-space edges that extend from islands have points of B_α on both sides, these free-space edges are not part of B_α and therefore the islands are not completely

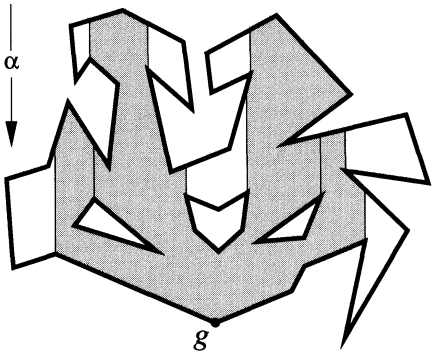


FIG. 3. *The α-backprojection B_α.*

surrounded by B_α.

These properties hold not only for the example, but also for the general case, as we show in Lemma 2.4. For that proof only, we use the following definition and lemma.

The *upper envelope* of an island $\mathcal{I}$ is the set of boundary points of $\mathcal{I}$ with no points of $\mathcal{I}$ above them (with respect to a direction α). We can classify points p of the upper envelope of an island $\mathcal{I}$ into three categories:

(i) *Left-inclined*: p is in B_α, and the robot slides left through p (α is pointing down).

(ii) *Uninclined*: p is not in B_α, or p is the goal g.

(iii) *Right-inclined*: p is in B_α, and the robot slides right through p.

LEMMA 2.3. *The upper envelope of an island $\mathcal{I}$ contains at least one uninclined point.*

Proof. Let l be the leftmost point of $\mathcal{I}$, and r be the rightmost point. Let C be the polygonal chain of the boundary of $\mathcal{I}$ counterclockwise from r to l, and let U be the upper envelope of $\mathcal{I}$. The points of U are a subset of the points of C, and form intervals along the chain C.

If l is uninclined, we're done; otherwise, l must be left-inclined. (If $\mathcal{I}$ contains g, then both l and r are uninclined.) If r is uninclined, we're also done, so r must be right-inclined. The points along U cannot change from left-inclined to right-inclined without having an uninclined point in between, and since neither category is empty, this proves the existence of an uninclined point. □

LEMMA 2.4. *The α-backprojection is a simply-connected subset of P whose interior is bounded by straight line segments. Each edge on the boundary of B_α is one of three types:*

- Bottom edges: *P-edges that have the interior of P locally above them*;
- Top edges: *portions of P-edges that have the interior of P locally below them*;
- Free-space edges: *edges that extend from a P-vertex upward all the way through the interior of P.*

Proof. The α-backprojection is connected because every point p in B_α has a path connecting p to g. We prove that B_α is simply connected by showing that every simple loop ℓ in B_α is contractible to a point.

Notice that every maximal vertical segment in P that intersects $\ell \subset B_\alpha$ is in B_α. If ℓ contains no islands then these vertical segments cannot end inside ℓ. Thus, ℓ can be contracted to a point in its interior, which is in B_α.

On the other hand, any simple loop ℓ that contains an island contains an uninclined point on the upper envelope of each island. These upper envelopes, being monotone curves with respect to $\alpha^{\perp}$, can be ordered consistently with aboveness in direction α. Then there is a point $p \in \ell$ that is directly above an uninclined point on the uppermost curve, so p is not in B_α.

Since every point $p \in B_\alpha$ implies that every vertical segment in P that passes through p is also in B_α, the free-space edges on the boundary of B_α must be parallel to α and extend all the way through the interior of P. The remaining edges of B_α must be P-edges; P-edges on which the robot slides are included in B_α in their entirety. □

Another way of looking at the structure of B_α is more suitable for computing it. Consider the partition of P into trapezoids using the α-visibility graph [14]. Lemma 2.4 implies that every trapezoidal face and every vertical edge is either completely included or completely excluded from B_α.

LEMMA 2.5. *Consider the directed graph $G = (V, E)$, where*

- *the nodes V consist of the trapezoids of the α-backprojection, plus the goal g, and*
- *there is an arc from u to v if points of v slide into u.*

Then G is a tree (rooted at g).

Proof. G is connected, by Lemma 2.4. Points of each trapezoid slide into exactly one other trapezoid (or into g), so G is either a tree or contains a directed cycle. If G contained a cycle, we could exhibit a loop trail, contradicting Lemma 2.2. □

The algorithms that we present in §§3 and 4 use a *triangulation* of the environment P, that is, a partition of the interior of P into triangles whose vertices are P-vertices [33]. The following lemma will be used to help bound the working storage used by these algorithms.

LEMMA 2.6. *For a fixed triangulation of the environment P, every triangulation edge intersects B_α in at most k segments. (Recall that k is the number of disjoint simple polygons describing the environment.)*

Proof. First we prove a slightly stronger claim for the simple polygon case ($k = 1$): if points p and q are in B_α, and the segment $\overline{pq}$ is in free space, then the whole segment $\overline{pq}$ is in B_α. Without loss of generality, assume that p is to the left of q. Let p' be the first boundary point that $T_\alpha(p)$ hits, and let q' be the first boundary point that $T_\alpha(q)$ hits. Let C be the portion of the boundary counterclockwise from p' to q'. The polygon Q bounded by $\overline{qp}$, $\overline{pp'}$, C, and $\overline{q'q}$ is completely contained inside the environment. Now, any point r on the upper envelope U of C is along $T_\alpha(p)$ or $T_\alpha(q)$ or both, since if there were a point r on U not on either trail, the ray from r in direction $-\alpha$ would break Q into two disjoint pieces with $T_\alpha(p)$ confined to one piece and $T_\alpha(q)$ confined to the other. Because both trails are known to reach the same point g, the claim follows.

We can reduce the general case in which the outside polygon contains $k-1$ islands to the simple polygon case. It follows from Lemma 2.4 that every island has a ray extending from the island through free space in direction $-\alpha$ that is not contained in the α-backprojection. If we cut tunnels in the environment along these rays, we connect all the islands to the outside polygon without changing the α-backprojection. These tunnels divide each triangulation edge into at most k segments, and by the claim above, each such segment intersects the α-backprojection in at most one interval. □

2.3. The noncrossing theorem. We now state the key property of compliant motion paths that holds whether or not the environment has islands, and for arbitrary friction cones. Given two directions α and β, the trails $T_\alpha(p)$ and $T_\beta(p)$ start together at p and may touch at other points but never cross. The following theorem establishes this result by extending one trail to cut the plane into two pieces and showing that the other trail remains in one piece.

THEOREM 2.7. *Let p be a point in the environment, and let α and β be directions. Then there is a biinfinite polygonal curve Γ that does not cross itself (it may touch itself without crossing), such that*

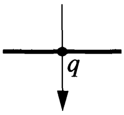
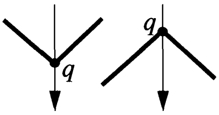
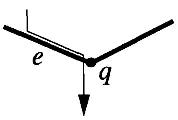
- *T_α is contained in Γ, and*
- *T_β is contained in the closure of exactly one of the open sets into which Γ partitions the plane.*

The proof is lengthy and only Theorem 2.7 itself is used elsewhere in this paper. It may be easier to skim this section at first reading and return to it after acquiring a better feel for compliant motion.

To construct Γ we concatenate *stabbing trails* $\widehat{T}_\alpha$ and $\widehat{T}_{-\alpha}$, where the stabbing trail $\widehat{T}_\gamma$ for direction γ is an infinite extension of the compliant motion trail T_γ defined in the next paragraph and Table 1.

For this section only, split every P-edge lengthwise into two—an *internal* and an *external* edge. Give external edges a friction cone that is a 180° rotation of the corresponding internal edge friction cone. This determines the behavior of a hypothetical robot hitting the P-edge from the environment exterior. (The real robot cannot do this, of course.) The stabbing trail $\widehat{T}_\gamma(p)$ starts off by tracing the compliant motion of a robot starting at point p with commanded direction γ. Whenever the compliant motion terminates at a sticky point q on a P-edge or a vertex, the stabbing trail proceeds as described in Table 1. (The first and second cases are relatively natural; we shall see later why the third case is different.)

TABLE 1
The stabbing trail $\widehat{T}_\gamma$ encounters a sticky point q.

Case	Example	Action
q is on a P-edge.		The motion switches sides between the interior and exterior sides of the P-edge.
q is a sticky vertex reached from free space.		We resume the motion at q. (Motion may resume in the interior or exterior.)
q is a sticky vertex reached by sliding along incident P-edge e.		We interrupt the sliding at a point q' along e that is infinitesimally close to q.[a] We then switch between interior and exterior at q', and continue.

[a] The stabbing trail $\widehat{T}_\gamma$ stops at point q' along e whose vertical projection (a projection on a line perpendicular to γ) is at a distance less than some δ from the vertical projection of q. A reasonable upper bound for δ is the smallest difference between the vertical projections of any two vertices whose vertical projections are distinct.

Observation 2.1 holds for stabbing trails, because a stabbing trail is nothing

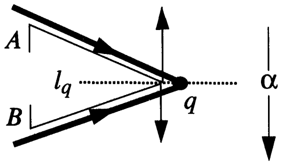


FIG. 4. *Stabbing trails do not meet.*

but a regular trail in a modified environment, in which the original P-edges are duplicated and infinitesimal holes are placed where a stabbing trail goes through a wall. Lemma 2.2 holds for stabbing trails for the same reason, applying the additional fact that a stabbing trail can slide only along one side (either the interior or the exterior) of any given edge. At some point, the stabbing trail crosses the boundary of the outside polygon for the last time and continues in a straight line to infinity.

The crux of the proof of Theorem 2.7 is the following lemma.

LEMMA 2.8. *The stabbing trails $\widehat{T}_{-\alpha}(p)$ and $\widehat{T}_{\alpha}(p)$ share no points other than p.*

Proof. We prove the lemma by assuming that $\widehat{T}_{-\alpha}$ and $\widehat{T}_{\alpha}$ do meet and by deriving a contradiction. Let q be the first point (after p) along $\widehat{T}_{-\alpha}$ that is also on $\widehat{T}_{\alpha}$. Denote the portion of $\widehat{T}_{-\alpha}$ between p and q by A, and the portion of $\widehat{T}_{\alpha}$ between p and q by B. Since q is the first meeting point, A and B share p and q, but no other points.

When A and B separate at p, A goes "up" and B goes "down." More precisely, there is a line σ_p through p (σ_p is not necessarily horizontal) such that in the vicinity of p, A is above σ_p and B is below σ_p. (We may take σ_p to be the angular bisector of the initial segments of A and B.)

Observation 2.1 applies to the partial trails A and B—namely, rightmost and leftmost points, r and l, on $A \cup B$ are p or q or appear on free-space edges going up on A or going down on B. We look at the cases for these extreme points and show that A and B must intersect. This contradiction establishes the lemma.

First, suppose that neither r nor l is the point q. If both r and l come from one trail, say A, then the portion of A between r and l separates p below from q above because A remains between r and l, goes up at both r and l, and has no loops. But B also remains between r and l and must therefore cross A to connect p to q. On the other hand, if r comes from A and l comes from B, then consider where q is with respect to the path π from r to l that follows A to p and B to l. If q is above, then B must cross π from below to reach q; if q is below, then A must cross π from above.

Next, suppose that $r = q$. An analysis like the one above shows that no matter whether the leftmost point l is on A, is on B, or is p, the path A must reach q from above and B from below. But then the trails $\widehat{T}_{-\alpha}(p)$ and $\widehat{T}_{\alpha}(p)$ must both be sliding towards q on edges that meet at a vertex at q. As illustrated in Figure 4, however, q is a sticky vertex for both trails, and the stabbing trails would pierce their respective edges according to line 3 in Table 1, before reaching q. This contradicts their supposed behavior. □

As we mentioned at the beginning of the section, we would like to use the concatenation of $\widehat{T}_{\alpha}$ and $\widehat{T}_{-\alpha}$ as our Γ. However, we want T_{α} to be completely contained in Γ, which is not the case if $\delta > 0$ (see footnote *a* at the bottom of Table 1). Thus we let $\delta \rightarrow 0$. This may cause Γ to become a nonsimple curve. Figure 4 shows that if $\delta = 0$, the two stabbing trails may touch each other at a sticky vertex and then separate. The following lemma helps characterize the points where Γ touches itself.

LEMMA 2.9. *Let γ be an arbitrary direction, and let v be a point on the boundary*

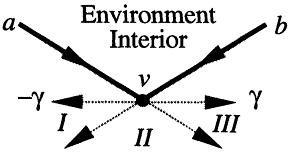


FIG. 5. *v is sticky for γ and $-\gamma$.*

of the environment that is sticky for both direction γ and direction $-\gamma$. Then v is sticky for any direction that reaches v (from another point in the environment).

Proof. Every P-edge can affect only one of the directions γ or $-\gamma$, and therefore v has to be a vertex. Furthermore, γ has to point into one of the P-edges incident to v, and $-\gamma$ has to point into the other P-edge. This leaves only one possibility, as shown in Figure 5: $\overrightarrow{av}$ is a conveyor belt at least strong enough to carry $-\gamma$ into v, and $\overrightarrow{bv}$ is a conveyor belt that carries γ into v.

Directions in zone I are forced by $-\gamma$ to slide along $\overrightarrow{av}$, and cannot slide out of v along $\overrightarrow{vb}$. Similarly, directions in zone III are forced by γ to slide along $\overrightarrow{bv}$ and are not affected by $\overrightarrow{av}$. Directions in zone II are forced by both P-edges to slide into v. Finally, any other direction that reaches v has to be either between $-\gamma$ and $\overrightarrow{va}$ but still slide towards v, or be between γ and $\overrightarrow{vb}$ but still slide towards v; in both cases the direction cannot slide out of v along the other P-edge. □

Lemma 2.9 implies that with Γ set to be the concatenation of the two stabbing trails $\widehat{T}_\alpha$ and $\widehat{T}_{-\alpha}$ (where $\delta = 0$), T_β can touch only one of the vertices where Γ touches itself (and terminate there), and T_β slides along at most one of the edges incident to that vertex.

LEMMA 2.10. *Let Γ be the concatenation of the stabbing trails $\widehat{T}_\alpha$ and $\widehat{T}_{-\alpha}$, with $\delta = 0$. The trail T_β never crosses Γ.*

Proof. When $\delta > 0$, the simply connected polygonal chain Γ divides the plane into two parts, the "left" and the "right" parts (when α points down). The left half is locally on the right of $\widehat{T}_\alpha$ (when looking in the direction of progress of $\widehat{T}_\alpha$), and locally on the left of $\widehat{T}_{-\alpha}$. When $\delta = 0$, there may be more than two regions. We label the regions "left" and "right" by continuity as $\delta \to 0$.

Looking along T_β, starting from p, we can see two kinds of "events," a *separation*—T_β separates from Γ (either at p, or after a common segment), and a *meeting*—T_β runs into Γ. The first event is a separation. Without loss of generality, we can assume that T_β goes to the right side of Γ at that point (so β is in the range counterclockwise from α to $-\alpha$). We prove the following three claims by induction on the events:

(1) All meetings occur along P-edges.

(2) If a point $q \neq p$ on a P-edge e is common to both T_β and Γ, then T_β slides along e through q in the same direction as Γ (recall that at any point, Γ is either $\widehat{T}_\alpha$ or $\widehat{T}_{-\alpha}$).

(3) At a separation, T_β always splits off to the right side of Γ (meaning locally to the left of $\widehat{T}_\alpha$ and locally to the right of $\widehat{T}_{-\alpha}$).

Base case. The first event: claims (1) and (2) are trivially true, and claim (3) is true by assumption.

Induction step. Suppose that all three claims are true for the portion of T_β between p and the jth event (the *old* event), and prove that the three claims hold for the portion between the jth and the $(j+1)$th event (the *new* event).

Case 1. The old event is a separation. We have to prove claims (1) and (2) for the new event in this case. By the induction hypothesis, T_β splits off to the right side of Γ. Since Γ goes to infinity at both ends, T_β stays on the right side until the new meeting. A free-space meeting cannot happen, since at a free-space meeting T_β crosses from the left side of Γ to the right side, so claim (1) is true at the new meeting: the meeting occurs at a point q along a P-edge e. At the meeting point q, exactly one of T_β or Γ reaches q from free space (one has to reach q from free space, or else it would not be an event; and by the argument of the previous sentence (the same argument that rules out free-space meetings), both cannot reach q from free space).

We divide Case 1 into the following subcases.

Case 1a. T_β reaches q from free space. First suppose that T_β hits $\widehat{T}_\alpha$. If T_α is sliding to the left (when α is pointing down) then T_β has to reach q from the left side of Γ, since β is counterclockwise from α (see Figure 6), but this is a contradiction of the induction hypothesis, so $\widehat{T}_\alpha$ slides to the right. Now, since α is in the right-sliding zone of edge e, and β is counterclockwise from α, β is also in the right-sliding zone. Similarly, we can prove that if T_β hits $\widehat{T}_{-\alpha}$, then $\widehat{T}_{-\alpha}$ has to be sliding right, which forces T_β to slide right, too.

Case 1b. T_β slides into q. This is essentially the same as Case 1a, except that α and β switch roles (β is counterclockwise from α).

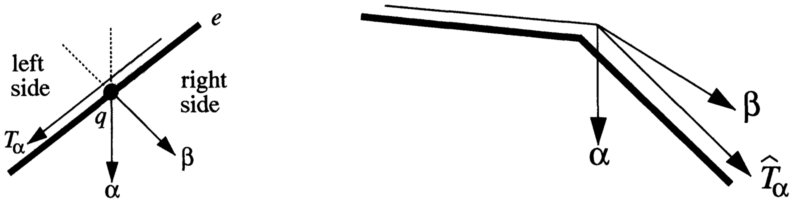


FIG. 6. *Trail* T_β *never crosses* Γ: *induction cases* 1a *and* 2b.

Case 2. The old event is a meeting. We have to prove claim (2) for the common portion of T_β and Γ, and claim (3) for the new event.

We prove claim (2) by a new induction on the number of edges in the common portion. The base case is by the main induction hypothesis of this proof. For the induction step, if T_β keeps on sliding along Γ, we use the fact that $\widehat{T}_\alpha$ continues to slide right and $\widehat{T}_{-\alpha}$ continues to slide left (see Case 1 above) to conclude that they still force T_β to slide along with them. We omit the details for brevity.

For claim (3), we have a few subcases.

Case 2a. Both trails go into free space. By our initial assumption, T_β splits off counterclockwise of $\widehat{T}_\alpha$ and clockwise of $\widehat{T}_{-\alpha}$, so in both cases T_β splits off to the right side of Γ.

Case 2b. T_β goes into free space. First suppose that prior to the separation, T_β was sliding together with $\widehat{T}_\alpha$. It is not hard to verify that the only way T_β can go into free space and leave $\widehat{T}_\alpha$ sliding is if both trails were sliding to the right. This way, the environment interior was on the left-hand side when looking down $\widehat{T}_\alpha$. Since T_β split into free space it must have turned to the left-hand side when looking down $\widehat{T}_\alpha$, which is the right side of Γ. (See Figure 6.) Similarly, when T_β leaves $\widehat{T}_{-\alpha}$, it turns to the right-hand side when looking down $\widehat{T}_{-\alpha}$, which is again the right side of Γ.

Case 2c. Γ goes into free space, but not through a puncture (see Table 1, line 3). This case is similar to] Case 2b.

Case 2d. Γ goes into free space through a puncture. Either T_β stops at the meeting point q, or it does not and only one of $\widehat{T}_\alpha$ and $\widehat{T}_{-\alpha}$ reaches q (Lemma 2.9). In the latter case T_β and $\widehat{T}_\alpha$ (or $\widehat{T}_{-\alpha}$) are sliding to the right (else T_β would stop at q). When $\widehat{T}_\alpha$ ($\widehat{T}_{-\alpha}$) goes through the puncture at q, T_β continues to travel right, and hence splits off to the right side of Γ. □

This concludes the proof of Theorem 2.7, the noncrossing theorem.

2.4. Properties of regions bounded by trails. It follows from Theorem 2.7 and Lemma 2.2 that if q is any common point of T_α and T_β other than p, then there are points $s_1, s_2, \ldots, s_j$ and $t_1, t_2, \ldots, t_j$, all common to T_α and T_β (it is possible that some of the s_i's coincide with some of the t_i's), such that

- the points appear in the order $s_1, t_1, s_2, t_2, \ldots, s_j, t_j$ along both T_α and T_β;
- T_α does not touch T_β between s_i and t_i, and coincides with T_β between t_i and s_{i+1};
- $p = s_1$ and $q = t_j$.

This means that the portions of T_α and T_β between p and q bound a simply connected polygonal region $R_{\alpha,\beta,q}(p)$. When α and β are both good directions (see §2.1) for some point p, we simply write $R_{\alpha,\beta}(p)$ instead of $R_{\alpha,\beta,g}(p)$; when p is fixed, we may write $R_{\alpha,\beta}$.

The trails T_α and T_β form the boundary of $R_{\alpha,\beta}$. In §4.1, we need to know more about whether T_α circumnavigates this region clockwise or counterclockwise. To answer this question, we need to define the direction 0 (zero) mentioned at the beginning of §2. The direction 0 is one that is not good for any point in the environment other than the goal itself. An example of such a direction is the one that points into the environment along the bisector of the two P-edges incident to the goal. This direction does not point into either P-edge incident to g, so a robot following commanded direction 0 cannot slide along them. Furthermore, in free space locally around g, it points away from g. Therefore, no compliant motion path with a programmed direction 0 can end at g. The following lemma states a consequence of the definition of direction 0.

LEMMA 2.11. *If $\alpha < \beta$ are good directions for p, then T_α traverses $R_{\alpha,\beta}$ counterclockwise (and T_β traverses it clockwise).*

Proof. Consider $\widehat{T}_0(p)$, the stabbing trail (defined in §2.3) starting at p with direction 0. Theorem 2.7 guarantees that $\widehat{T}_0(p)$ does not cross either T_α or T_β. Furthermore, by definition, $\widehat{T}_0(p)$ does not reach g. Since $\widehat{T}_0$ is an unbounded path, it cannot be inside $R_{\alpha,\beta}$. It follows that wherever T_α and T_β separate for the first time (usually at p), $R_{\alpha,\beta}$ lies in the interval counterclockwise of T_α and clockwise of T_β. This proves the lemma. □

Combining the above properties, we conclude the section with the following theorem.

THEOREM 2.12. *Let p be a point inside P, and let $\alpha < \beta$ be good directions for p. Also, assume that $R_{\alpha,\beta}$ contains no islands of P. Then every $\gamma \in [\alpha, \beta]$ is a good direction for p. Furthermore, if $[\alpha, \beta]$ is smaller than* 180°, *then every $\gamma \in [\alpha, \beta]$ is a good direction for any $q \in R_{\alpha,\beta}$.*

Proof. We prove the theorem in three steps: first, we prove that every $\gamma \in [\alpha, \beta]$ is good for p; second, we prove that if $[\alpha, \beta]$ is smaller than 180°, then α is good for any point q along $T_\beta(p)$ and β is good for any point q along $T_\alpha(p)$; third, we prove

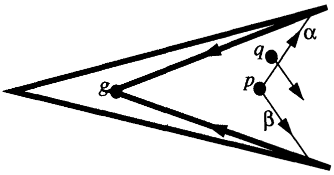


FIG. 7. *q may escape if* $[\alpha, \beta] > 180°$, *but not if it is less.*

that if $[\alpha, \beta]$ is smaller than 180°, then both α and β are good directions for every point q inside $R_{\alpha,\beta}$. The third step gives a polygon $R_{\alpha,\beta}(q)$; applying the first step to it proves the second part of the theorem.

Let γ be a direction in $[\alpha, \beta]$. By Lemma 2.11, T_γ starts off by going into $R_{\alpha,\beta}$. A case analysis similar to that in the proof of Lemma 2.10 shows that any time T_γ hits either T_α or T_β, T_γ slides along with them. Since the interior of $R_{\alpha,\beta}$ is free of P-edges, T_γ cannot get stuck in the interior of $R_{\alpha,\beta}$. Because compliant motion trails inside a bounded region are of finite lengths (in fact, if the absolute height of the region with respect to α is h, and there are n P-edges totalling length l, the trail T_α cannot exceed $nh + l$ in length), and because the trail T_γ cannot get stuck inside or on the boundary of $R_{\alpha,\beta}$, it follows that T_γ has to reach g.

Continuing with the second part of the proof, let q be a point along $T_\alpha(p)$. We prove the stated claim by induction on the number of times $T_\beta(q)$ hits $T_\alpha(p)$. In the base case, if $T_\beta(q)$ does not hit $T_\alpha(p)$, then $T_\beta(q)$ must hit $T_\beta(p)$—this is because $R_{\alpha,\beta}$ is bounded, and $T_\beta(q)$ splits off to the left of $T_\alpha(p)$ and therefore into $R_{\alpha,\beta}$ ($[\alpha, \beta]$ is smaller than 180°), and the interior of $R_{\alpha,\beta}$ is free of P-edges. Once $T_\beta(q)$ hits $T_\beta(p)$, it follows $T_\beta(p)$ all the way to g. For the inductive step, look at the first time $T_\beta(q)$ hits $T_\alpha(p)$. This hitting point has to be along a P-edge, and since $T_\alpha(p)$ slides on that P-edge, $T_\beta(q)$ slides there also. Follow the two trails together until the first time they separate, and we've reduced the number of meetings between $T_\beta(q)$ and $T_\alpha(p)$. The other direction of the claim is symmetric.

For the third part of the proof, let q be an arbitrary point strictly inside $R_{\alpha,\beta}$. Since $R_{\alpha,\beta}$ is bounded, when we extend a ray from q in direction $-\alpha$, we hit the boundary of $R_{\alpha,\beta}$, say at point r. This point cannot be along $T_\alpha(p)$: if r belonged to $T_\alpha(p)$ then so would q, but we have assumed that q is inside $R_{\alpha,\beta}$. Therefore, r must lie along $T_\beta(p)$. Applying the previous step, we conclude that α is good for r, and therefore good for q. □

Remark. If $[\alpha, \beta] > 180°$, then the second part of Theorem 2.12 does not hold. Figure 7 shows a counterexample. In the figure, the two P-edges incident to the goal are conveyor belts toward the goal. Direction β is bad for the point q.

3. The simple polygon algorithm. In this section we restrict ourselves to the case $k = 1$, namely, to environments that have no islands. We present an algorithm for single-step, perfect-control, perfect-sensing compliant motion planning for an environment with n vertices and a goal vertex g. The algorithm spends $O(n \log n)$ preprocessing time to produce a linear-size data structure that represents the nondirectional backprojection, which is the union of the α-backprojections for all α. This data structure subsequently enables us to produce a "single-step plan" (i.e., report all the good directions), once the starting point p is given, in $O(\log n)$ time.

Theorem 2.12 says that if P contains no islands, then the good directions of every point p inside P form a single range, bounded by the directions *start*(p) and *stop*(p).

This range is empty for points that cannot reach the goal in a single compliant motion. If we can compute the functions *start*(p) and *stop*(p) for any given point p, we can easily answer queries for the one-step compliant motion planning problem.

In this section, we show how to compute two subdivisions of P, called the *start* and *stop* subdivisions, that correspond to the two functions. Each subdivision partitions the nondirectional backprojection into $O(n)$ triangles and trapezoids. Each triangle or trapezoid groups together points for which we compute the corresponding function in a uniform way. By performing a point location in each subdivision, we can in $O(\log n)$ time locate the region that contains a query point, and then compute the value of the corresponding function in constant time. Once a triangulation of P is given, we compute the subdivisions in linear space and $O(\tau \log n)$ time, where τ is the number of triangulation edges that intersect the nondirectional backprojection. (We can compute a triangulation of a simple polygon in $O(n \log n)$ time [17], or in linear time by a more complex algorithm [6].)

In §3.1, we describe an algorithm that computes the α-backprojection: the set of all points for which α is a good direction. This is easily done by sweeping through the trapezoidation given by the α-visibility map; we show how to use an extension of the *path hull* data structure of Dobkin et al. [7] to perform the sweep without the trapezoidation. Path hulls are described in detail in §6. This algorithm is used as a subroutine in constructing the nondirectional backprojection.

Sections 3.2 through 3.4 describe and analyze the rotation algorithm for computing the start and stop subdivisions that represent the nondirectional backprojection. The idea is to maintain the α-backprojection B_α as α rotates.

Most of the time (informally speaking) we can rotate α slightly without changing B_α significantly: only the free-space edges bounding B_α rotate. We distinguish two types of free-space edges: the *leading* free-space edges sweep over new points and the *trailing* free-space edges sweep over points that leave the α-backprojection as α turns counterclockwise. For some α's, which we call *events*, the structure of B_α may change more dramatically. Events occur when a rotating free-space edge hits a vertex, a P-edge starts or stops being slidable, or an edge turns over and goes from top to bottom. We maintain the directions of events in a priority queue. While the queue is not empty, the algorithm rotates the current direction α until just past the first scheduled event. Processing the event involves updating B_α and the list of events. Again, path hulls are used to detect events.

3.1. Computing the α-backprojection . If we decompose P into trapezoids with sides parallel to α by cutting through P at every vertex [32], then the α-backprojection consists of a subset of adjacent trapezoids that form a tree. (See Lemma 2.5.) We can compute this by sweeping free-space edges, which are the boundaries of trapezoids. Starting from the edge into the goal g, we sweep over adjacent trapezoids whose bases can slide into g.

Because a triangulation can be converted into a trapezoidation in linear time [14], we could also use a trapezoid algorithm to compute an α-backprojection given a triangulation. (A different approach is developed by Heffernan and Mitchell [21], who obtain an α-backprojection in linear time without using a trapezoidation or triangulation.)

We can also compute backprojection B_α by a sweep algorithm based on a triangulation of P rather than on a trapezoidation. The primary advantage of this algorithm is that it allows us to consider rotating α without the need to maintain a trapezoidation parallel to α. It also can be slightly more efficient—it can be implemented to

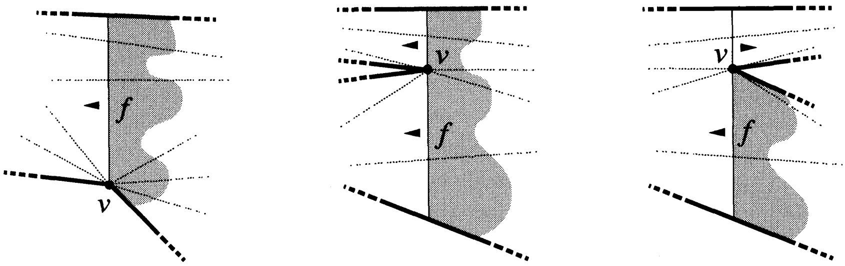


FIG. 8. *f encounters a vertex v at its base, middle, or top.*

compute B_α in time proportional to the number of triangles that intersect B_α, which we denote τ. (Because we are anticipating the rotation algorithm, we describe an implementation that is suboptimal for computing B_α by a factor of $O(\log n)$.)

Let us assume that α is directed vertically downward for the description of this algorithm. We begin our sweep by erecting two free-space edges, one moving left and one moving right, by walking upward from the goal g through triangles of P.

To sweep, consider how a free-space edge f that is moving left might encounter a vertex v: either at its base, in the middle, or at the top, as illustrated in Figure 8. Edges moving right are handled in a symmetric fashion.

If f encounters v at its base and the P-edge counterclockwise (ccw) from f slides into v, then f sweeps over v to the next vertex, adding a trapezoid to B_α. If f encounters v at its middle, then we split f into two segments. The lower one continues to sweep left to the next vertex it hits. The upper segment sweeps left if and only if the P-edge ccw from f is slidable into v. Finally, if f encounters v at the top, then we have two cases. If one of the P-edges incident to v goes to the left, then f continues to sweep left. If both P-edges incident to v go right, then we extend f beyond v until it hits the polygon boundary. We create a new front segment for the upper portion of f which sweeps right if the upper P-edge incident to v is slidable into v; segment f continues to sweep left.

Without the trapezoidation, determining the next vertex that the free-space edge f hits is nontrivial. We define $L(f)$ to be the chain of left endpoints of the triangulation edges that intersect f. The order of the points along $L(f)$ is the order in which the corresponding edges cross f. Similarly, we define $R(f)$ to be the chain of right endpoints. Because the triangles that intersect f form a simple polygon, $L(f)$ and $R(f)$ are simple polygonal chains. The following lemma uses these chains to characterize the first vertex hit by a free-space edge.

LEMMA 3.1. *Let f be a free-space edge that touches P-edges t and b, which extend to the left of f, and let H be the convex hull of $L(f)$. No polygon vertex lies in the region bounded by f, t, b, and H. A similar claim holds for $R(f)$.*

Proof. Suppose that a vertex v lies strictly inside the region. At least two triangulation edges are incident to v; these edges cannot cross other triangulation edges or f. If the edges incident to v all intersect the convex hull H, however, then v is a reflex vertex. Since a triangulation has no reflex vertices, no vertex exists in the region. □

Suppose that the free-space edge f moves left. (Moving right is handled symmetrically.) The first vertex v that f hits is the tangent to the hull H of $L(f)$. The algorithm maintains H using the *path hull* data structure that is described in detail in §6. Here it suffices to know that the structure maintains the hull of a polygonal chain and supports, in amortized logarithmic time, finding tangents to the hull, adding and

TABLE 2
Changing $L(f)$ when f hits a vertex v.

Pos. of v	Action
Base of f	Delete v from $L(f)$ and add the left endpoints of edges that have v on their right.
Middle of f	Split $L(f)$ into two v, remove v, and add to each chain the left endpoints of edges that have v as their right endpoint.
Top of f	Remove v from $L(f)$ and add the left endpoints of the edges that have v on their right. If f extends above v, we add the left endpoints of the new edges that cross f. We also generate a new free-space edge f' that has v as its base.

deleting vertices from the ends of the chain, and spliting a chain or merging two subchains. (The split/merge bound does not allow intermixed sequences of splits and merges, but the algorithm does not require them.) All the path hulls in use at any time during the algorithm take only $O(n)$ storage altogether.

LEMMA 3.2. *Assuming that the appropriate path hull has been computed for a free-space edge, one can add trapezoids to the backprojection in amortized $O(\log n)$ time per intersected triangle.*

Proof. Let us go back to Figure 8, this time paying attention to the triangulation edges that cross f. When a leading free-space edge f slides to the left, the changes to $L(f)$ depend on the position of v along f, as described in Table 2.

We do similar operations to $R(f)$ for trailing free-space edges that slide to the right. Whenever a free-space edge moves and its chain changes, we compute the next event associated with it by updating the convex hull of the chain and finding the appropriate tangent to it. This takes logarithmic time per triangle intersected by B_α. □

This shows how to carry out the sweep, which will also be used as a subroutine in the rotation algorithm of the next section.

3.2. Events for the rotation algorithm. As mentioned at the beginning of §3, the core of the preprocessing phase is a rotation algorithm, which maintains the α-backprojection B_α while rotating α through 360°. In this section we describe the representation of B_α, the events at which it changes, and how to handle these events.

The rotation algorithm represents B_α as a sequence of edges and vertices. We define $seq(B_\alpha) = (a_1, a_2, \ldots, a_k)$ as follows. Initially, we set $seq(B_\alpha)$ to $()$, the empty sequence. Walking around B_α counterclockwise, starting and ending at the goal vertex g, we append an element to $seq(B_\alpha)$ for each P-edge and for each P-vertex at the base of a free-space edge. For a P-edge e along the boundary of B_α, we append either $\underline{e}$ or $\overline{e}$, depending on whether e is a top or a bottom edge; for a P-vertex v at the base of a free-space edge, we append v. (If a bottom edge meets a top edge at a vertex, we introduce a null free-space edge at that vertex.) The polygon B_α, the trapezoid tree of B_α, and the pair $\langle \alpha, seq(B_\alpha)\rangle$ are all linear-time equivalent representations for the α-backprojection. We define events as those directions α for which $seq(B_{\alpha^-}) \neq seq(B_{\alpha^+})$. The next lemma characterizes events.

LEMMA 3.3. *A direction α is an event if and only if one of the following conditions holds at this direction* (*see Figure* 9):

- *a bottom edge of B_α ceases to be slidable, or a P-edge next to the base of a free-space edge becomes slidable into B_α (a "sliding event");*
- *a free-space edge encounters a P-vertex (a "visibility event");*

- *a top edge becomes a bottom edge, or a bottom edge becomes a top edge (a "turnover").*

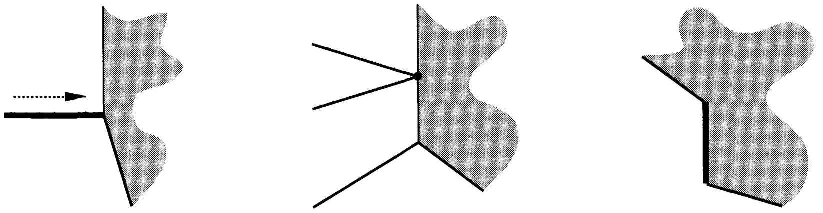


FIG. 9. *Three rotation events:* a sliding event, *a* visibility event, *and a* turnover.

Proof. If one of these events occurs, the backprojection gains or loses a vertex of the polygon, or a P-edge moves between the top and bottom of B_α. In either case $seq(B_\alpha)$ changes.

Conversely, suppose that we rotate α so that none of these conditions occurs—we argue that $seq(B_\alpha)$ does not change. Consider maintaining the vertical (that is, α) visibility graph as well as B_α. While the vertical visibility graph is unchanged, the trapezoid algorithm includes the same set of trapezoids in the backprojection (because no sliding events occur), so the sequence does not change. When the visibility graph changes, α is parallel to the line through two vertices. By assumption, the lower vertex is not the base of a free-space edge, so the higher vertex does not enter or leave the backprojection as we rotate; the vertices are not endpoints of a single P-edge, so no P-edge moves between the top and bottom of B_α. Thus $seq(B_\alpha)$ does not change. □

The rotation algorithm maintains the current direction α, the sequence $seq(B_{\alpha^+})$, and for each free-space edge f, a path hull for the left endpoints of triangulation edges that intersect f, as described in Lemma 3.1. Splits will be performed on path hulls for leading edges and merges on path hulls for trailing edges.

The algorithm uses a priority queue to determine the next event. The queue contains, for each P-edge in B_α or adjacent to it, the next direction at which the edge becomes or ceases to be slidable (for sliding events) and the direction (i.e., angle) of the P-edge itself (for turnover events). The queue also contains, for each free-space edge, the angle at which the edge hits the next vertex as α rotates (for visibility events). It follows from Lemma 3.3 that the event β at the top of the priority queue satisfies the condition $seq(B_{\alpha^+}) = seq(B_{\beta^-}) \neq seq(B_{\beta^+})$. At every step, the algorithm takes the next event off the top of the priority queue and computes the changes that have to be made to $seq(B_{\alpha+})$ when α advances to β. In the process, the algorithm may discover and enqueue further events. The algorithm may also recognize that some events in the queue have become obsolete and dequeue them.

We now describe a single iteration of the rotation algorithm. The sequence changes during the transition from α^- to α^+ for the event α. Since α is fixed throughout the rest of the section, we define direction α to be vertically downward.

A turnover event does not involve any changes to the α-backprojection, and the changes to the sequence can be carried out in constant time. We also create or destroy a null free-space edge at the point where a top edge touches a bottom edge.

There are two kinds of events in which points are *added* to B_α: a sliding event involving a P-edge adjacent to the base of a leading free-space edge f, or a visibility event involving a leading free-space edge f. In both cases, we first rotate f to angle α and then call on the procedure of §3.1 to add trapezoids to B_α that can now be reached because of the event on f. This may cause the scheduling of visibility events for new free-space edges, and turnover and sliding events for new P-edges.

There are two kinds of events that can cause B_α to *lose* trapezoids: a sliding event involving a bottom edge inside the α-backprojection, and a visibility event involving a trailing free-space edge. If we explicitly had a trapezoidation of B_α, then in both cases we would need to cut a subtree off the trapezoid tree. Since we do not have the trapezoidation, we need to run the algorithm of §3.1 in reverse to remove trapezoids from B_α.

In the case of a sliding event, suppose that bottom edge e just stopped being slidable towards its vertex u. We walk away from u on edges that slide towards u until we reach the base of a free-space edge f—all edges that we walk on will be removed from B_α. Then we rotate f to angle α, and begin sliding f in the direction of u (which is to the left, since f was a trailing edge). Suppose that f encounters a vertex v (refer back to Figure 8 for the three cases). First, a base vertex v is easily handled as before. Second, a vertex v in the middle of f implies that anything that slides into v can no longer reach the goal. We process the upper P-edge into v as if it had a stop sliding event, moving free-space edges in until there is one based at v. Then we destroy that free-space edge, shorten f to end at v, and continue sliding f. Third, a vertex v at the top of f is easy to handle unless both incident P-edges go right. In that case, we essentially trigger a stop sliding event for the upper P-edge into v. When the portion of the backprojection that slid to v has been removed and there is a free-space edge based at v, then we merge that edge with f and continue sliding f.

In the case of a visibility event for a trailing free-space edge f, we rotate f to angle α, remove regions from B_α, if necessary, as described in the previous paragraph, construct the new free-space edge f, and continue.

In scheduling new visibility events, the algorithm must detect the first vertex hit by a rotating free-space edge. This can be done easily using path hulls as in Lemma 3.1. Each free-space edge represents the right and left chains by a path hull. However, only one of these chains is maintained per edge at any given moment: When a leading free-space edge is born, we generate its left chain. When a trailing free-space edge is born, we generate its right chain and use it to move the edge (and split it, if necessary, as described in §3.1) until we cannot move any further, at which time we disassemble its right chain and generate its left chain. When a leading free-space edge has to move to the right, we disassemble its left chain and generate its right chain. The following lemma shows that left and right motions are not intermixed. Thus we create the path hulls for $L(f)$ and $R(f)$ only once.

LEMMA 3.4. *A leading edge can move right only once, just before it disappears. A trailing edge can move right when it is created, but never again.*

Proof. If a leading edge moved to the right and then rotated counterclockwise, or if a trailing edge rotated and then moved right, then some points would enter the α-backprojection a second time, in violation of Theorem 2.12. □

3.3. The start and stop subdivisions.

The start and stop subdivisions encode the first and last angles for which points can reach the goal. Each subdivision consists of trapezoids and triangular sectors whose union is exactly the nondirectional backprojection.

Let p be a point in P. For the start subdivision, if p is in a trapezoid with parallel sides of direction α, then $start(p) = \alpha^+$; if p is in a sector and sees the base vertex of the sector in direction α, then $start(p) = \alpha^+$. (See Figure 10.) For the stop subdivision, the corresponding direction is α^-. (In the special case in which the goal g is the base vertex of the sector containing p, then $start(p) = \alpha$ (or $stop(p) = \alpha$).)

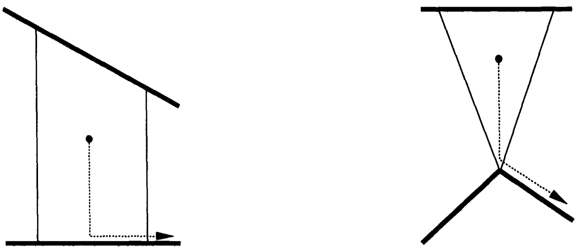


FIG. 10. *Determining start angles from trapezoids or sectors of the start subdivision.*

Whenever the rotation algorithm moves a free-space edge, it adds regions to the start and/or stop subdivisions. When a leading edge slides left, or when a trailing edge slides right, it sweeps over trapezoids that are added to the start subdivision. Similarly, when a leading edge slides right or when a trailing edge slides left, it sweeps over trapezoids that go into to the stop subdivision.

If we rotate the direction α counterclockwise and the sequence $seq(B_\alpha)$ does not change, then each leading edge f sweeps out a triangular sector of points and adds them to B_α. A point p in the sector defined by f first enters the α-backprojection at the direction of the ray from p to the base of f. Trailing edges remove points from B_α. We include the sectors swept by leading edges in the start subdivision; those swept by trailing edges go into the stop subdivision.

In addition to α and $seq(B_\alpha)$, we maintain, for each free-space edge f, the direction of the edge at the last event involving f. Before the rotation algorithm moves a free-space edge laterally, we check if the current α is different from the direction associated with the free-space edge. If so, we "rotate" the free-space edge, adding the triangular sector to the appropriate subdivision.

Some start sectors may be generated by stop events (visibility events involving a trailing free-space edge, or stop-sliding events). This happens when a leading free-space edge disappears from the α-backprojection at a stop event, since we first create a start sector for such an edge.

3.4. Analysis of the algorithm. We now show that the running time of the rotation algorithm is $O(\tau \log n)$, where τ is the number of triangles that intersect the nondirectional backprojection.

First, we show that $O(\tau)$ path hull operations are performed. When a triangulation edge enters or leaves the α-backprojection, we add its endpoints to or delete them from at most two hulls. Each vertex that enters the backprojection can cause a split; each one that leaves can cause a merge. By Theorem 2.12, a vertex enters at most once and leaves at most once. Thus the total number of hull operations is $O(\tau)$; they take $O(\tau \log n)$ amortized time. As for space, there are $O(\tau)$ path hulls at any given moment, and each path hull can be of size $O(n)$; nevertheless, the total size of all the path hulls is $O(n)$ (this follows from Lemma 2.6). We explain in §6 a technique in which we allocate a fixed block of storage of size $O(n)$ for all the path hulls, and hence the working storage is $O(n)$.

Second, the total time spent on priority queue operations is also $O(\tau \log n)$. Since each P-edge enters and leaves the backprojection once, we perform a constant number of priority queue operations for each P-edge. Each time a hull changes, we compute a new tangent and add it to the queue; there are $O(\tau)$ such operations.

Finally, since each subdivision is composed of $O(\tau)$ triangles and trapezoids, we can preprocess them for point location in $O(\tau)$ time [10, 24]. At query time, we use two $O(\log \tau)$ point locations to find the regions of the start and stop subdivisions that contain the query point p. Using these regions we can determine the start and stop angles in constant time.

THEOREM 3.5. *Given a triangulated simple polygon P and a goal vertex or edge, we can build a data structure to support queries for one-step compliant motion inside P under perfect control. Queries take $O(\log \tau)$ time, preprocessing takes $O(\tau \log n)$ time and $O(n)$ space, and the data structure takes $O(\tau)$ space, where τ is the number of triangles that intersect the nondirectional backprojection.*

4. Polygons with islands. Now we consider an environment P with $k-1$ islands ($k \geq 1$), as described in §2.1. In this section, we show how to modify the simple polygon algorithm to handle the islands. The modified algorithm spends $O(kn \log n)$ time preprocessing the n-vertex environment P and the goal vertex g to produce a data structure with size $O(kn)$; at query time, the algorithm reports all the good directions for a given starting point p in $O(k \log n)$ time.

For any given α, the α-backprojection has the same structure as in the simple polygon case (since Lemma 2.4 does not rely on P being a simple polygon), namely, it has bottom, free-space, and top edges, and it can be computed from the trapezoids of the α-visibility graph. Also, the events that change the structure of B_α, represented by $seq(B_\alpha)$, are still the same, as stated in Lemma 3.3. Thus, we can use the rotation algorithm that was defined in §3 to maintain the B_α. The problem is that the good directions of a point p inside P may form several ranges, and hence as we rotate α, points may enter and leave B_α more than once. This means that the subdivisions of §3.3 are not subdivisions any more, but rather "piles" of possibly overlapping triangles and trapezoids. In order to build an efficient query structure for polygons with islands, we will separate each pile into layers so that every layer is indeed a subdivision.

While partitioning the trapezoids and triangles into layers leads to an efficient query structure, there is a less reductionistic way to think about the nondirectional backprojection. Triangles and trapezoids of the start subdivision can be stitched together along their boundaries where a free-space edge moves from one region to another to form a simply connected surface (due to Lemma 2.2) in space that only overlaps itself when projected to the plane. Thus, the start triangles and trapezoids give a subdivision of a subset of the universal covering space of the polygon [22]—the stop triangles and trapezoids give an alternate subdivision of the same subset. We concentrate on layers in this section, but the surface model simplifies some of the applications in §5.

To analyze the complexity of this amended algorithm, we need to bound the number of events that take place as we rotate α, the total size of the piles, and the storage requirements of the path hulls used during the rotation. The following lemma bounds the number of times a point can enter and leave the α-backprojection.

LEMMA 4.1. *For a given point p, there are at most k ranges that define all the good directions for p.*

Proof. Let the smallest good direction for p be α, and the largest good direction for p be ω. If the good directions for p form l ranges, then there are $l-1$ "bad" ranges between α and ω. Every such bad range must contain an island by Theorem 2.12. An island can appear in at most one bad range, since it can never be split by a trail.

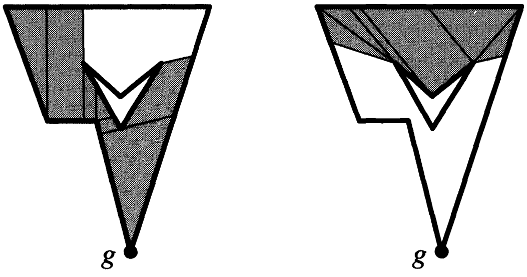


FIG. 11. *Layers of the start subdivision.*

Therefore, $l - 1$ cannot exceed the number of islands, which is $k - 1$. □

We proceed as follows: in §4.1 we describe the layering scheme of the preprocessing phase and prove its correctness. In §4.2 we describe how to answer queries for the single-step, perfect-control, perfect-position sensing case using the data structures produced by the preprocessing phase. (Section 5 shows how to answer more realistic queries.) Finally, we analyze the complexity of this scheme in §4.3.

4.1. Layers for the start and stop subdivisions. Let us limit our attention to the pile that contains trapezoids and triangles that are *added* to B_α, known as the "start pile" (the treatment of the stop pile is symmetric). We need to specify the layering scheme that we use to partition this pile into subdivisions.

We use k layers, one for each of the k boundary components of the environment. Recall that a (nonturnover) event involves adding a triangular sector and zero or more trapezoids. The base vertex of a triangular sector belongs to a particular boundary component; we place the sector in the layer of that component. All the trapezoids go to the layer of the component where the *event* took place, as follows:[1]

- for a sliding event, when an edge next to B_α becomes slidable, we use the boundary component that contains that edge;
- for a visibility event, when a leading free-space edge encounters a vertex, we use the boundary component that contains the base vertex of the free-space edge.

Figure 11 illustrates the layering scheme for a small example with friction cones that are the normals to the edges and goal vertex at the bottom of the environment. On the left is the layer that corresponds to the outside polygon, and on the right is the layer that corresponds to the island. Note that some trapezoids that are based on the island go into the layer of the outside polygon, because the event in which they were created took place on the outside polygon. In order to prove the correctness of this scheme, we need to show that all the regions in any layer are disjoint. First we need the following lemmas.

LEMMA 4.2. *Suppose that p enters the α-backprojection at α^+ during an event associated with boundary component C, and suppose that $\beta > \alpha^+$ is also a good direction for p. Then C is outside $R_{\alpha^+,\beta}$.*

Proof. We know that $p \notin B_\alpha$ and $p \in B_{\alpha^+}$, so T_{α^+} has almost the same prefix as T_α, but T_α either gets stuck or takes a wrong turn at a point q on C. The point q is arbitrarily close to a point r on C where T_{α^+} goes through. (See Figure 12.)

Now, since we know that T_{α^+} goes counterclockwise around $R_{\alpha^+,\beta}$, the point q cannot be inside $R_{\alpha^+,\beta}$. Hence the whole interior of C must lie outside $R_{\alpha^+,\beta}$. □

[1] We must treat sectors and trapezoids differently because some sectors in the start pile can be generated by stop events, as mentioned in §3.3.

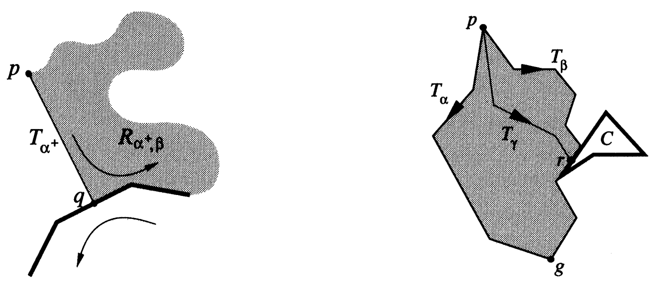


FIG. 12. *Proving Lemmas* 4.2 *and* 4.3.

LEMMA 4.3. *Let p be a point in the environment, let $\alpha < \beta$ be good directions for p, and let C be a boundary component such that $R_{\alpha,\beta}$ does not contain C. Then for any $\alpha < \gamma < \beta$, the trail T_γ cannot get stuck on C at any point that is not g, the goal vertex.*

Proof. By Theorem 2.7, T_γ stays inside $R_{\alpha,\beta}$. The only way T_γ can get stuck on C is if C touches $R_{\alpha,\beta}$ where T_γ gets stuck. Suppose that T_γ gets stuck at a point r on the boundary of $R_{\alpha,\beta}$. This means that T_γ cuts $R_{\alpha,\beta}$ into two pieces: piece A is bounded by T_α and T_γ near p, and piece B is bounded by T_γ and T_β near p. If $r \neq g$, only one of these two pieces contains g. (See Figure 12.)

Suppose that A contains g. Again by Theorem 2.7, T_β stays in B up to r. But T_β does not get stuck at r, so it crosses to the "left" (to A) at that point. Because T_γ always slides left more easily than T_β, it cannot get stuck at r.

The other case is symmetric. □

Using these lemmas, we can now prove the correctness of the layering scheme.

THEOREM 4.4. *For any layer, any two regions are disjoint.*

Proof. We prove the theorem by contradiction. Suppose that in the layer of boundary component C there are two regions that have a nonempty intersection, and suppose that p is in their intersection. One region implies α^+ as a start direction for p, and the other region implies β^+, where $\beta > \alpha$.

By Lemma 4.2, C is not contained in R_{α^+,β^+}. Since $\alpha^+ < \beta < \beta^+$, by Lemma 4.3 T_β cannot get stuck on C. However, p is in the region of β^+ because T_β *does* get stuck on C. This is a contradiction, and therefore no such point p exists. □

4.2. Queries in layers. When we are given a query point $p \in P$, we locate it in each layer of the start pile, and in each layer of the stop pile. Every region that contains p implies a corresponding direction in which p enters or leaves the α-backprojection (see §3.3), so when we sort the list of directions we end up with an alternating list of start–stop directions for p. This list specifies the good ranges for p, and this is the answer to the query.

4.3. Analysis. There are two quantities to consider: space and time. To help bound working storage, we have the following lemma.

LEMMA 4.5. *The total size of all the path hulls at any given α is $O(kn)$.*

Proof. The total size of the path hulls is the sum, over all the free-space edges, of the number of triangulation edges and P-edges that cross each edge. Reversing the order of summation, this number is equal to the sum, over all the triangulation edges and P-edges, of the number of free-space edges that the edge crosses. There are $2n + 3(k-2)$ triangulation edges and P-edges, and by Lemma 2.6, each such edge crosses at most $2k$ free-space edges. □

At any time during the algorithm, $O(kn)$ storage is needed for path hulls. Because individual path hulls may change their storage needs, statically allocating $O(kn)$ memory is not sufficient. Section 6 describes a dynamic storage allocation scheme that uses $O(kn \log n)$ storage altogether.

Obtaining the triangulation of a polygon with islands takes $O(n \log n)$ time [33, pp. 230–234] instead of the linear time possible in the simple polygon case. However, the total preprocessing time is dominated by the total number of trapezoids and sectors plus the number of turnover events (which do not contribute any regions to the pile, but have to be processed nevertheless), multiplied by the time that it takes to process each one.

LEMMA 4.6. *The total number of sectors and trapezoids in the start and stop piles is $O(kn)$.*

Proof. Each vertex of P enters B_α at most k times as α rotates. Every trapezoid or sector added to the start pile can be associated with one of these vertex entrances, and each vertex entrance has at most one trapezoid and one sector associated with it. A symmetric argument applies to the stop pile. □

Every P-edge can turn over at most twice, so there are $O(n)$ turnovers. Processing a trapezoid, a sector, or a turnover involves a path hull operation, and since the size of each path hull is $O(n)$, each operation takes $O(\log n)$ time. Therefore, the total preprocessing time is $O((kn + n)\log n) = O(kn \log n)$.

It is easy to keep track of the adjacency of the regions that we generate in each layer (since α rotates monotonically). All the regions in each layer are connected by adjacency to the island that is associated with that layer, so if we triangulate the island and add the triangles to the layer, we get a connected subdivision at no extra cost. We can preprocess this subdivision for point location using time and space linear in the size of the layer [10, 24].

Query answering involves $O(k)$ point location queries, which take $O(k \log n)$ total time, and sorting at most $2k$ numbers, which takes $O(k \log k) = O(k \log n)$ time.

THEOREM 4.7. *Given a triangulated polygonal environment P with n vertices and k islands, and a goal vertex or edge, we can build a data structure to support queries for one-step compliant motion inside P under perfect control. Queries take $O(k \log n)$ time, preprocessing takes $O(kn \log n)$ time and space, and the data structure takes $O(kn)$ space.*

5. Extensions to the main algorithm.

The algorithms of §§3 and 4 create data structures to answer queries for the perfect-control, perfect-position sensing case. However, once we have formed these structures, we can easily answer more realistic queries, allowing imperfect-control and imperfect-position sensing.

5.1. Imperfect control.

The control uncertainty of a robot is an angular constant ϵ in the range $0 \leq \epsilon < 90°$. If the robot is programmed to move in direction α, its actual direction of motion (in free space) at any given instant lies in the range $[\alpha - \epsilon, \alpha + \epsilon]$. The path the robot follows now is not unique, although to simplify our proofs we will assume that it is piecewise differentiable. We say that a direction α is *good* for a starting point p if every path the robot may take, when commanded to move from p in direction α, is guaranteed to get the robot to the goal. We relate perfect and imperfect control in Lemma 5.2, but first we need the following lemma.

LEMMA 5.1. *Let γ be a direction, and call it "down." Given a control uncertainty $\epsilon < 90°$, let $\widetilde{T_\gamma}(p)$ be any imperfect-control trail starting at p with commanded direction γ, and let q be a point along $\widetilde{T_\gamma}(p)$ such that no sliding occurs between p and q. Let* *the length of $\widetilde{T_\gamma}(p)$ between p and q be L, and let the vertical distance between p and q be D. Then there is a constant c_ϵ, which depends only on ϵ, such that $L \le c_\epsilon D$.*

Proof. Since $\epsilon < 90°$, and no sliding occurs between p and q, that portion of the trail $\widetilde{T_\gamma}(p)$ is monotone with respect to γ. Therefore, the trail $\widetilde{T_\gamma}(p)$ can be described by a continuous, piecewise differentiable function $f(x)$, where x measures the vertical distance from p, and $f(x)$ measures the horizontal displacement (positive values to the right) from p. We know that

- $f(0) = 0$;
- $|f'(x)| \le \tan\epsilon$ almost everywhere.

Now we can bound the length L of $\widetilde{T_\gamma}(p)$ as follows:

$$L = \int_0^D \sqrt{dx^2 + df^2} = \int_0^D dx\sqrt{1 + |f'(x)|^2} \le \int_0^D dx\sqrt{1 + \tan^2\epsilon} = \frac{D}{\cos\epsilon}.$$

We set c_ϵ to $1/\cos\epsilon$, and the lemma follows. □

We are now ready to prove the lemma relating perfect and imperfect trails.

LEMMA 5.2. *Let p be in P, and let (α, β) be one of the (at most k) ranges of good directions for p for a robot with perfect control. Let the control uncertainty be $\epsilon < 90°$.*

- *If (α, β) is wider than 2ϵ, then any $\gamma \in (\alpha + \epsilon, \beta - \epsilon)$ is a good direction for the imperfect robot;*
- *if (α, β) is not wider than 2ϵ, then no $\gamma \in (\alpha, \beta)$ is a good direction for the imperfect robot.*

Proof. Let γ be a direction in the range $(\alpha+\epsilon, \beta-\epsilon)$. It follows from Theorem 2.12 that for any point r in $R_{\gamma-\epsilon,\gamma+\epsilon}$, the range $(\gamma - \epsilon, \gamma + \epsilon)$ is a good range under the perfect control model. Now, let $\widetilde{T_\gamma}$ be an imperfect control trail starting at p with commanded direction γ. We can prove by a case analysis similar to that of Lemma 2.10 that whenever $\widetilde{T_\gamma}$ touches the boundary of $R_{\gamma-\epsilon,\gamma+\epsilon}$, it cannot cross outside of this region ($\widetilde{T_\gamma}$ cannot touch a free-space boundary edge; when $\widetilde{T_\gamma}$ touches a P-edge on the boundary, it is forced to slide in the same direction as the corresponding perfect-control trail; and when it separates from the boundary, it goes into free space inside the region).

We have to show that the robot indeed makes progress on its way to the goal. First, we consider the cases in which $\widetilde{T_\gamma}$ touches the boundary of $R_{\gamma-\epsilon,\gamma+\epsilon}$. Clearly, the places where $\widetilde{T_\gamma}$ touches $T_{\gamma-\epsilon}$ are consistently sorted along both trails (because if r occurs before s along $\widetilde{T_\gamma}$, then $R_{\gamma-\epsilon,\gamma+\epsilon}(r)$ contains $R_{\gamma-\epsilon,\gamma+\epsilon}(s)$). Now, by the proof of Theorem 2.12, when $\widetilde{T_\gamma}$ travels along a portion of the boundary of $R_{\gamma-\epsilon,\gamma+\epsilon}$, it slides in the same direction (never gets stuck, and never slides the wrong way, because both would imply that there is a direction in the range $(\gamma - \epsilon, \gamma + \epsilon) \subseteq (\alpha, \beta)$ that is not good for a point along the boundary of the region). Finally, whenever $\widetilde{T_\gamma}$ goes into free space, it has to hit the boundary again (this is because if r occurs before s along a free-space segment of $\widetilde{T_\gamma}$, then the area of $R_{\gamma-\epsilon,\gamma+\epsilon}(r)$ is strictly smaller than the area of $R_{\gamma-\epsilon,\gamma+\epsilon}(s)$). Since the lengths of $T_{\gamma-\epsilon}$ and $T_{\gamma+\epsilon}$ and the area of $R_{\gamma-\epsilon,\gamma+\epsilon}$ are all finite, $\widetilde{T_\gamma}$ has to reach g eventually.

Finally, we have to prove that we do not run into "Zeno's paradox," namely, that we do not have an infinitely long $\widetilde{T_\gamma}$. We use Lemma 5.1 to bound the length L of $\widetilde{T_\gamma}$ as follows:

$$L \le c_\epsilon D + B + c_\epsilon B,$$

where D is the distance in direction γ from p to g, and B is the total length of the boundary of $R_{\gamma-\epsilon,\gamma+\epsilon}$ (that is, the sum of the lengths of $T_{\gamma-\epsilon}$ and $T_{\gamma+\epsilon}$). The first term in the bound accounts for the free-space distance; if $\widetilde{T_\gamma}$ goes farther than this in free space, it overshoots the goal. The second term accounts for the sliding. Finally, the third term accounts for the fact that some of the edges on which $\widetilde{T_\gamma}$ slides on may act as conveyor belts, and carry the robot back up a distance that the robot may later travel through free space. Hence the length of $\widetilde{T_\gamma}$ is finite, and this concludes the proof of the first part of the lemma.

As for the second claim, by definition, α and β are not good directions for p. If (α, β) is not wider than 2ϵ, then for any γ in (α, β) it is possible for the robot to follow T_α as its imperfect control path for the commanded direction γ, and therefore γ is not a good commanded direction for the imperfect robot. □

Using Lemma 5.2, it is clear how we can use the structure from §4 to answer queries under the imperfect control model. First, we find the good ranges under the perfect control model, as described in §4. Next, we increase every start direction by ϵ and decrease every stop direction by ϵ. Every range that stays nonempty is a good range under the imperfect control model. Note that we use ϵ only at query time, so the preprocessing is independent of the control uncertainty. This means that the same data structures can support several robots, with different capabilities, that operate in the same environment.

5.2. The boundary of the nondirectional backprojection. Recall that the *nondirectional backprojection* is the set of all points that can reach the goal in a single compliant motion. If the robot has perfect control, this set is the union of all the regions in the start pile (which is identical to the union of all the regions in the stop pile). When the control uncertainty ϵ is greater than 0, the nondirectional backprojection is the set of all points that has at least one good range of size greater than 2ϵ.

We compute the boundary of the nondirectional backprojection. Specifically, we compute a closed curve ν whose interior, as defined by its winding number, is exactly the nondirectional backprojection. The curve ν is a simple Jordan curve on the universal covering space of the interior of P [22]. That is, the interior of ν always lies to the left of ν when ν is traversed counterclockwise, and ν contains no island of P in its interior. If $k = 1$ (the simple polygon case), then the universal covering space is just P itself, and ν is a simple curve made up of line segments and circular arcs. If $k > 1$, ν is not necessarily simple when projected down from the universal covering space to the plane.

The algorithm for computing ν is the same whether or not $k = 1$: we simply trace ν through the universal covering space. That is, we follow ν through the regions of the start and stop piles simultaneously. Whenever ν leaves one triangle or trapezoid, it enters an adjacent one, according to the adjacency relation when the two regions were created. The triangles and trapezoids of the start (stop) pile, stitched together by adjacency, form a subpolygon of the universal covering space.

To trace ν, we start a point p at g and move p counterclockwise along ν, walking through the start and stop piles simultaneously. At each step, the regions of the start and stop pile containing p dictate the line segment or circular arc that p must follow until it leaves one of the regions, as described below. The tracing algorithm adds this segment or arc to ν as it moves p. When p leaves a region, it crosses a free-space edge into an adjacent region.

Here are the three possibilities for the regions containing p, and the actions that the tracing algorithm takes in order to proceed to the next step. In each case, p moves so that the region locally to the left of ν has a good range larger than 2ϵ, as determined by the current regions of the start and stop piles, and the region locally to the right of ν does not (perhaps because ν follows the polygon boundary).

- If p lies in a triangular sector in both the start and stop pile, then ν follows a circular arc, the curve described by the apex of a fixed 2ϵ angle whose two rays pass through two fixed points (the sector bases). If the arc hits the boundary of P, ν follows that boundary, as appropriate.
- If p lies in a sector in one pile and in a trapezoid in the other, then ν follows a radial line of the sector; ν may also follow the boundary of P if necessary.
- If p lies in two trapezoids, then ν follows the boundary of their intersection.

The tracing algorithm needs to go across free-space edges of the piles, which may involve changing layers. We can perform this operation in constant time if, during the preprocessing phase, we record the adjacency information of the pile regions as follows. The algorithm creates a region in either pile by rotating a free-space edge or by sweeping a free-space edge across a trapezoid. We maintain, for each free-space edge, a pointer to the region ρ of the α-backprojection immediately neighboring it; we use this pointer to doubly link ρ with the new region over which the free-space edge sweeps, and then update the pointer to point to the new region.

The tracing algorithm takes time proportional to the number of times ν passes from one pile region to another. The following lemma shows that this quantity is $O(kn)$.

LEMMA 5.3. *Let ν be the boundary of the nondirectional backprojection in the imperfect-control case. Then the tracing algorithm follows ν through each region of either pile at most four times.*

Proof. The start and stop piles are both subdivisions of a single connected portion of the universal covering space of the interior of P [22]. Viewing the piles from the perspective of the universal covering space, we see that two regions in the start and stop piles intersect if and only if they have a common point p and the trails from p to g determined by the two regions are homotopic (they wind around the islands the same way).

The tracing algorithm traces ν through the universal covering space; that is to say, the tracing algorithm is purely local, following the region adjacencies determined when the regions were created.

To enter or leave a region of either pile, ν must cross a free-space edge of the region. Since each pile subdivides part of the universal covering space, the regions of the other pile that intersect the edge are all adjacent.

Let p be a point on a free-space edge of the start pile that lies inside ν. Let α be the direction determined by the free-space edge. By Theorem 2.12 and Lemma 5.2, the direction $\alpha + \epsilon$ is good for every point in $R_{\alpha,\alpha+2\epsilon}(p)$, in particular for every point on the free-space edge between p and the base of the edge, and hence all those points lie inside ν. Thus ν intersects each free-space edge at most twice, possibly at the endpoints. Each pile region has at most four free-space edges, so the lemma follows. □

5.3. Uncertainty in initial-position sensing. When the robot has imperfect-position sensing, we may know its starting position only approximately. Formally, we know that the robot is somewhere in a (typically small) connected subset S of the environment, known as the *start region*. In this case, we need to find directions that are good for all the points in S simultaneously. Note that the answer to a query may be "unreachable" even when every point in S can reach the goal.

LEMMA 5.4. *Let S be a connected subset of the environment. If a direction α is good for all the points on the boundary of S, then α is good for all points in S.*

Proof. Let p be a point in S. If p is not a boundary point, then $T_\alpha(p)$ hits the boundary of S. □

The lemma means that we can find maximum and minimum angles by looking at the boundary of S. We use the following technique to find the intersection of the good directions of all the points along the boundary of S. First, we locate a point p on the boundary of S in each of the two piles. The good ranges of p correspond to at most k pairs of start/stop regions that intersect in the universal covering space as in §5.2. For each such pair, we trace the boundary of S in the start and stop subdivisions of the universal cover simultaneously, maintaining the interval between the "largest" start-angle α and the "smallest" stop-angle ω. We stop a tracing when we arrive back at p (and we abort a tracing, discarding the corresponding interval, if it reaches a free-space edge that bounds the nondirectional backprojection). When we have completed all $O(k)$ tracings, we have generated the $O(k)$ good ranges for all the points on the boundary of S, and therefore all the points of S, under perfect control. We can shrink each such range by 2ϵ, as explained in §5.1, in order to obtain the result under control uncertainty.

Answering the initial query for p takes $O(k \log n)$ time, plus time proportional to the number of subdivision regions traversed during the $O(k)$ traces along the boundary of S, since we never visit the same region more than the number of times the boundary of S crosses it, and we spend constant time per triangle or trapezoid. For many "standard" starting regions, such as a disc (intersected with the environment) or a polygon with a constant number of sides, the number of triangles or trapezoids that intersect S is $O(kn)$ in the worst case, but is typically less.

5.4. Sensorless robots. We have so far implicitly assumed that the robot can detect the goal when it reaches it. If this is not the case (the robot is *sensorless* [9, 8, 12]), we can still use our data structures to answer queries.

A sensorless robot cannot detect the goal g, and so it must stick at g when it reaches it. This restricts the range of directions the robot can use; we denote the allowable range by $[\alpha, \omega]$. The range $[\alpha, \omega]$ contains the directions for which the robot cannot slide away from the goal on one of its incident edges. To answer a query for a sensorless robot with control uncertainty ϵ, with or without position uncertainty, we apply the appropriate algorithm described above, then intersect the query result with $[\alpha + \epsilon, \omega - \epsilon]$.

Once again, we can compute the boundary of the nondirectional backprojection for sensorless robots: we trace along the boundary while holding an angle fixed. Whereas in §5.2 the angle was the start–stop range, here it is the intersection of that range with $[\alpha, \omega]$.

5.5. Convex polygonal goal regions. By minor modifications to the rotation algorithm, we can handle convex polygonal goal regions instead of single vertices or edges. We leave more general goal regions as an open problem.

To compute the α-backprojection of a convex goal polygon G we could decompose the difference $P \backslash G$ into trapezoids by the α-visibility map and then traverse trapezoids starting with the set of trapezoids whose bottom edges are on G. (See §3.1.) To apply the rotation algorithm to maintain this α-backprojection, we need to add *turnover events* for the edges of G to the priority queue. By a slight abuse of notation, we say that a bottom-to-top turnover occurs when the rotating direction α goes from negative

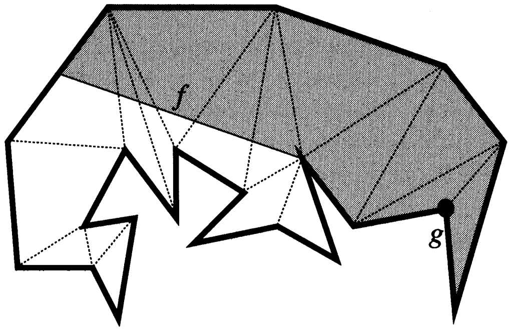


FIG. 13. *The free-space edges bounding the nondirectional backprojection cutoff bays.*

to positive projection on the normal to an edge of G; a top-to-bottom turnover occurs when the projection goes from positive to negative. (See §3.2.)

Since G is assumed convex, top-to-bottom turnovers occur only at the leftmost edge, with α vertically downward. Suppose that b was the leftpoint point of G until the turnover for edge (a, b) made a the leftmost point. We handle this turnover as if it were a sliding event from b to a—we create a sector for the free-space edge that was based at b, slide the free-space edge to a, and add the left endpoints of triangulation edges whose right endpoint is a to the path hull for the free-space edge. Bottom-to-top turnovers occur only at the rightmost edge and can be handled by deleting triangulation edges from the path hull. The analysis of §4.3 still holds if n is changed to the total number of vertices in G and P.

5.6. The multistep problem. A robot in an arbitrary polygonal environment may not be able to reach a goal g in a single step (this corresponds to the fact that there may be points p for which no direction is good). In this section, we consider the multistep problem for the following cases:

- A simple-polygon environment ($k = 1$), perfect control and goal sensing. We modify the algorithm of §3 to handle multistep planning within the same time and space bounds, namely, $O(n \log n)$ preprocessing time, linear-size data structures, and $O(\log n)$ query time. One can easily see that in this case, a robot can always reach g in n steps.
- Arbitrary k, sensorless (with control uncertainty), and with plans restricted to be of the form $(g_0, \alpha_1, g_1, \alpha_2, g_2, \ldots, \alpha_m, g_m)$, where $g_0 = p$, $g_m = g$, g_i is either a vertex or an edge, and any point in g_{i-1} reaches g_i via α_i. Our solution runs in $O(kn^2 \log n)$ preprocessing time, $O(kn^2)$ space, and $O(kn \log n)$ query time.

Consider the multistep problem inside a simple polygon under perfect-control and perfect-goal sensing. Suppose that we computed a nondirectional backprojection B from g that did not include the whole polygon P. Since B is composed of triangles and trapezoids bounded by P-edges and free-space edges, the unreached portions of P must be *bays* bounded by free-space edges, as illustrated in Figure 13.

A robot starting inside a bay bounded by a free-space edge f escapes the bay if it reaches f. We would like to run the rotation algorithm on the bay with f as the goal. However, we must first fix up the triangulation at the mouth of the bay. We need to triangulate the polygon defined by f and its left chain (or right chain). This polygon is weakly visible from f, and hence it can be triangulated in linear time by a simple algorithm [36].

Note that all the points in the triangles cut by f can reach f. At the next step, these triangles are strictly inside the nondirectional backprojection of f and thus can never be cut again. We run the algorithm of §3 on each bay, perhaps forming smaller bays. We repeat the process until every point is in some backprojection. Since the rotation algorithm runs in $O(\tau \log n)$ time, where τ is the number of triangles that intersect the backprojection, and every triangle is in at most two bays, the total time is only $O(n \log n)$.

With each trapezoid and triangle in the start and stop subdivisions, we store the free-space edge that is the subgoal and the number of steps to reach g. After preprocessing for point location, we can, in $O(\log n)$ time, answer queries about the number of steps a point requires to reach g. If the robot is capable of stopping when it reaches a free-space edge f, then we can list the directions and goals for the entire path by looking at the direction and goal of each successive edge subgoal.

Solving the multistep-planning problem with goal sensing and imperfect control is complicated by the fact that the boundary of the one-step nondirectional backprojection contains circular arcs. For sensorless robots with imperfect control, however, we can still say something about the multistep planning problem. Since the robot is sensorless, we take all subgoals to be vertices or edges—after the first step, the robot moves from one vertex or edge of P to another. This suggests the following graph-based approach. In $O(kn^2 \log n)$ time and $O(kn^2)$ space, compute the nondirectional backprojection of each P-vertex and each P-edge. Now we have $O(n)$ start and stop piles, and we can answer queries with each P-vertex or P-edge as a goal. Using these piles, we construct a directed graph. The nodes are the P-vertices and P-edges, and there is an arc from v_1 to v_2 if and only if v_1 can reach v_2 by a single step. For each goal v_2, we compute all arcs into it in $O(kn)$ total time, visiting each region in the start and stop piles for v_2 a constant number of times. We perform a breadth-first search on the graph starting from g, marking every node with its distance from g in the graph. We use this graph to solve the multistep-planning problem for sensorless robots. Given a query point p inside P, we find a P-vertex or P-edge that is reachable from p and is closest to g in the graph by performing at most n queries. This takes $O(kn \log n)$ time.

Note. In this case g does not have to be fixed: we simply defer the breadth-first search to the query phase, so the total time required for a query increases to $O(n^2 + kn \log n)$.

6. The path hull data structure. In this section we describe path hulls, the convex hull representation that the rotation algorithm of §§3 and 4 uses to find the nondirectional backprojection in $O(kn \log n)$ time. These path hulls are an extension of a data structure due to Dobkin et al. [7]. A *path hull* represents the convex hull of a simple polygonal path π, and consists of two separate *semipath hulls*; the path π is the concatenation of two portions, and each semipath hull represents the convex hull of one of the two portions. The convex hull representation is simple: each semipath hull stores the vertices of the convex hull of its portion in an array. This array supports binary search, and hence supports logarithmic-time intersection finding and tangent finding on the hull.

Path hulls support the operations required by §§3.1 and 3.2. In particular, for a path π with m vertices and convex hull h, a path hull that stores π and represents h supports the following operations:

1. Find tangents to h, either from a point outside h or parallel to a given line, in $O(\log m)$ time.

2. Add a vertex to either end of the path π (as long as the resulting path is simple) and update the hull in $O(\log m)$ time.
3. Delete a vertex from either end of the path and update the hull in constant amortized time, but worst-case $O(m)$ time.
4. Split π at a vertex to get two subpaths, each including the splitting vertex, and compute the convex hulls of each subpath in $O(m)$ worst-case time.
5. Merge two paths, disjoint except for a single vertex, into a single path of m vertices, and compute its convex hull in $O(m)$ worst-case time.

The worst-case time bounds quoted above are not particularly attractive; what makes path hulls useful is the amortized time bounds on intermixed sequences of these operations. In particular, if we perform an intermixed sequence of $O(m)$ operations of types 1, 2, 3, and 4 on paths with a total of m vertices, the total time required is $O(m \log m)$. The same bound holds for an intermixed sequence of $O(m)$ operations of types 1, 2, 3, and 5. We can obtain these amortized bounds because we use a pair of semipath hulls to represent each path; we balance the lengths of the two portions of the path to bound the average complexity of the operations.

The remainder of this section gives the details of path hulls. We first describe and analyze semipath hulls, from which we build path hulls. Next we tell how to implement path hulls based on the underlying semipath hull structures, characterizing precisely the operations that path hulls support. Finally, we use this characterization to prove the amortized time bounds quoted above. We defer until §6.4 a description of how to allocate storage for semipath hulls in our backprojection computation.

In what follows, we speak of simple paths as being directed. If π is a simple path, we distinguish between its endpoints by calling one end the *anchor* of the path and the other end the *free end* of the path. The path is directed from the anchor to the free end.

6.1. Semipath hulls. The semipath hull of a simple polygonal path represents the convex hull of the path by storing the hull vertices in a linear array; it allows vertices to be added to or deleted from the free end of the path. We use the notation $\mathit{SPH}(\pi, x)$ to denote the semipath hull of a simple path π anchored at x. (We have changed the nomenclature of Dobkin et al. slightly: what we call semipath hulls were called path hulls in that paper [7].)

For a simple path π with anchor x and free end v', the counterclockwise sequence of the vertices on its convex hull is a subsequence of $v' \ldots x \ldots v'$, the concatenation of the path and its reversal. The semipath hull stores the sequence of hull vertices in a deque (double-ended queue). The convex hull vertex closest to the free end of the path, call it v, appears at both ends of the deque. The subsequence property ensures that adding vertices to or deleting vertices from the free end of the path affects only the ends of the deque.

When vertices are added to or deleted from the free end of the path, the semipath hull updates the deque. The update method is based on Melkman's incremental algorithm for finding the convex hull of a simple path [29], which we now describe. Each step of Melkman's algorithm adds a new vertex to the simple path, then finds the convex hull of the resulting path. As above, let v be the convex hull vertex closest to the old free end, and let w be the new free end being added. If w lies inside the angle formed by the two hull edges incident to v, then it lies inside the convex hull, as in Figure 14. If w lies outside the old hull, the algorithm uses a Graham scan [18] to pop zero or more vertices off each end of the deque, then pushes the new vertex onto both ends of the deque.

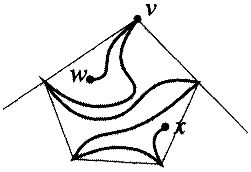


FIG. 14. *Testing if w is inside or outside the previous hull.*

A semipath hull consists of two data structures: a deque and a "transcript" stack. The deque contains the vertices of the convex hull, as described above, and the transcript stack records the push and pop operations needed to construct the current deque from scratch when path vertices are added one at a time. To add a vertex to the path, we use Melkman's algorithm to update the deque, and then record the update operations on the transcript stack. To delete a vertex, we play the transcript backwards: we pop operations off the transcript stack and perform their inverse operations until the deque reaches the state that existed before the vertex was added.

Allocating storage for the semipath hull is easy if we know a priori the maximum number of vertices that will ever belong to the path. If the upper bound on the number of vertices in the path is m, then we allocate an array of $2m - 1$ memory locations for the deque. We put the anchor vertex at the middle of the array; because there are at most $m - 1$ pushes on either end of the deque, the deque cannot overflow its array. Similarly, we allocate a block of $O(m)$ storage for the transcript stack, since the total number of operations to create the deque is at most $4m$.

The approach described so far may spend $O(m)$ time per vertex addition or deletion, since either operation may require $O(m)$ deque pops or pushes. We can avoid this overhead by using the array to do lazy pops. When we add a new vertex w, we find the tangents from w to the old convex hull using increasing-increment binary search (see below) on the array. This tells us which vertices Melkman's algorithm would pop off the deque. Instead of popping the vertices explicitly, we change the array indices that delimit the ends of the deque, as if the vertices had been popped off. When we push w onto the ends of the deque, the transcript remembers the vertex that was overwritten. To delete w and restore the deque to its former state, we simply replace w with the vertex that it overwrote, then change the deque-delimiting indices back to their previous values. Because implicitly popped array locations are altered only by reversible pushes, no information is lost during restoration. By using lazy popping, we reduce vertex addition time to $O(1)$ plus the time needed to find the tangents, and reduce vertex deletion time to $O(1)$.

We use a variant of binary search to find the tangents from the point w to the old hull. Ordinary binary search on a hull with k vertices takes $O(\log k)$ time. By using "increasing-increment" search, we reduce the search time to logarithmic in d, where d is the number of vertices that are popped off the hull by the addition of w. This form of binary search borrows the fundamental idea of finger search trees [23], but incurs little of the complexity of that data structure. We search in the array for the tangent vertices, starting at the ends of the deque (at v) and working toward the middle. Consider the search inward from the left end of the deque. By performing a constant-time test, we can tell whether the tangent vertex lies in the array interval between a query vertex and the left end of the deque. We start with the query vertex adjacent to the left end of the deque, then double the distance (measured in array indices) between the query vertex and the left end of the deque until the tangent vertex lies in the interval. We then use ordinary binary search on the interval to find the tangent vertex. The cost of the search is proportional to the logarithm of the interval size, that is, the logarithm of the number of vertices to be popped off on the left end of the deque. If d is the total number of vertices popped off by the addition of w, the cost of adding w is $O(\log(d+2))$. (The extra 2 allows for the case of no pops at all.)

Because a semipath hull stores the convex hull vertices in an array, it supports binary search algorithms to find tangents and intersections in logarithmic time. Thus we have established the following theorem.

THEOREM 6.1. *Let π be a directed simple path that is subject to addition and deletion of vertices at the free end of the path. If it is known beforehand that π will never have more than m vertices, then we can represent the convex hull of π in a semipath hull that supports* (1) *finding tangents to the hull, either from points outside the hull or parallel to given lines, in $O(\log m)$ time*; (2) *vertex addition at the free end of the path in $O(\log(d+2))$ time, where d is the number of vertices the new vertex removes from the convex hull*; *and* (3) *vertex deletion from the free end in constant time.*

6.2. Path hulls. We now discuss path hulls and their use of semipath hulls. A path hull for a path π distinguishes some vertex x of π. The vertex x divides π into two subpaths π_1 and π_2; the path hull stores $SPH(\pi_1, x)$ and $SPH(\pi_2, x)$. We use the notation $PH(\pi, x)$ to denote the path hull for π with distinguished vertex x. To make deletions at both ends of π easy, we want x to be close to the middle of π. However, repeated deletions or additions at one end may unbalance the path hull. We use amortization arguments (Theorems 6.2 and 6.3) to control the cost of imbalance.

At the beginning of §6 we summarized the operations that path hulls support. Now let us describe those operations in more detail, using Theorem 6.1 to bound their running times. We use h, h_1, and h_2 to denote the convex hulls of π, π_1, and π_2, and m to denote the number of vertices of π.

1. To find the tangents to h, either from a point outside h or parallel to a query line, we find the tangents to h_1 and h_2 separately, then choose the tangents to h from these four tangents. This takes $O(\log m)$ time.
2. To add a vertex to π, we add it to the semipath hull for π_1 or π_2, as appropriate. This takes $O(\log(d+2))$ time, where d is number of vertices popped from the appropriate semipath hull deque.
3. To delete a vertex w from π, we would like to delete it from exactly one of π_1 and π_2. If w is not equal to x, the middle vertex of the path hull, then deletion takes $O(1)$ time. If $w = x$, then we must rebuild the path hull, since one of the semipath hulls does not support deleting x. Suppose that π_1 is x. Then we find the middle vertex of $\pi_2 = \pi$, call it z, and build the path hull $PH(\pi, z)$, constructing the two semipath hulls by adding vertices. It takes $O(m)$ time to disassemble $SPH(\pi_2, x)$ by deleting all its vertices and then to build the two new semipath hulls. (The cost of building is $O(\sum_i \log(d_i+2))$, where d_i is the number of vertices popped off at the ith step of construction. Because $\sum_i d_i$ is at most $2m$, the construction cost is $O(m)$.) Disassembly and reconstruction also require $O(m)$ temporary storage for the vertices of π.
4. To split π at a vertex $y \in \pi_1$, we delete vertices of π_1 until y becomes the free end of π_1. Let the deleted path be π_1', let its length be m_1', and let its

midpoint be z. We build the path hull $PH(\pi_1', z)$ as in the previous case. These operations take $O(m_1')$ time and temporary storage.

5. To merge two paths π_1 and π_2, we disassemble one path hull and add it to the other. Suppose that π_1 and π_2 are disjoint except for a single vertex. Let the lengths of π_1 and π_2 be m_1 and m_2, and without loss of generality assume that $m_1 \leq m_2$. We disassemble the path hull for π_1 by deleting all its vertices, then add the vertices to the path hull for π_2. This takes $O(m_1)$ time for the disassembly and $O(m_1)$ temporary storage. The time for reassembly is $O(\sum_i \log(d_i + 2))$, as in 3 above, but now $\sum_i d_i \leq 2(m_1 + m_2)$, and so the total time for merging is $O(m_1 + m_1 \log(m_2/m_1)) = O(m_1 \log(1 + m_2/m_1))$.

6.3. Analysis. Now that we have characterized the operations that we perform on path hulls, we use the characterization to bound the time required by sequences of the operations. We consider intermixed sequences of operations 1, 2, 3, and 4 or 1, 2, 3, and 5. Our time bounds are amortized: we bound the time for the whole sequence of operations, but not for individual operations. The proofs use *potential functions* to amortize the costs of the various operations; that is, we prove that for each operation, the actual running time plus the change in the potential function is $O(\log m)$. Because the potential function for sequences that include splits is different from the one for sequences that include merges, we cannot give a good bound for the time required by an intermixed sequence that includes both splits and merges. An alternating sequence of m splits and merges may take $\Theta(m^2)$ time.

THEOREM 6.2. *Given a collection of path hulls and singleton vertices that contain a total of m vertices, we can perform an intermixed sequence of $O(m)$ tangent computations, vertex additions, vertex deletions, and path splits in $O(m \log m)$ total time.*

Proof. The preceding characterization of path hull operations gives the asymptotic complexity of each. To bound the complexity of a sequence of such operations, we must be more exact in our accounting. We assign a precise cost to each operation, then define a potential function related to the costs. We define the cost of a tangent computation to be $\log m'$ for a path hull with m' vertices. The cost of a vertex addition is $\log(d+2)$, where d is the number of vertices popped off the semipath hull by the new vertex. The cost of a deletion is either 1 in the easy case, or m' in the case when a path hull with m' vertices must be rebuilt. The cost of a split is k, where k is the size of the path hull that is rebuilt.

We define a potential function on the collection of path hulls. For a single path hull $PH(\pi, x)$, let m' be the length of π and let m_1 and m_2 be the lengths of the two portions of π produced by cutting at x, so $m_1 + m_2 = m' + 1$. We define the potential of $PH(\pi, x)$ to be

$$2m' \log m' + |m_1 - m_2|.$$

The first term of the potential is used to pay for splitting costs, and the second to pay for rebalancing when necessary. The coefficients are chosen to balance the two terms against each other for the splitting operation. The potential of the whole collection of path hulls, which we call Φ, is the sum of the individual potentials. The *amortized cost* of an operation is its true cost plus the change in potential, $\Delta\Phi$. We show that the amortized cost of each operation is $O(\log m)$. Notice that Φ is nonnegative and bounded above by $2m \log m + m$; if we start with a collection of path hulls whose potential is nonzero, we are still assured that the total time spent on a sequence of $O(m)$ operations is $O(m \log m)$.

For a tangent computation, $\Delta\Phi$ is zero, and so the amortized cost is the same as the true cost, $\log m' = O(\log m)$.

For a vertex addition to a semipath hull with m' vertices, $\Delta\Phi$ is at most

$$\begin{aligned}&2(m'+1)\log(m'+1) - 2m'\log m' + 1\\ &\quad = 2\log(m'+1) + 1 + 2m'\log(1+1/m')\\ &\quad = 2\log m' + O(1).\end{aligned}$$

The amortized cost is $O(\log m)$.

For a vertex deletion there are two cases. If no rebuilding is needed, then $\Delta\Phi$ is at most

$$\begin{aligned}&2(m'-1)\log(m'-1) - 2m'\log m' + 1\\ &\quad = -2\log(m'-1) + 1 + 2m'\log(1-1/m')\\ &\quad = -2\log m' + O(1)\\ &\quad = O(1).\end{aligned}$$

The amortized cost is certainly $O(\log m)$. If the path hull must be rebuilt, then the second term of the potential becomes significant. The actual cost is m', and $\Delta\Phi$ is $-2\log m' + O(1) - m'$, so the amortized cost is the same as in the first case.

For a split, the actual cost is k, where k is the size of the path hull that must be rebuilt. There are two cases depending on whether k is more or less than half of m', the length of the path being split. If $k \le m'/2$, then the first term of the potential gives what we want: $\Delta\Phi$ is at most

$$\begin{gathered}2(m'-k+1)\log\frac{m'-k+1}{m'} + 2k\log\frac{k}{m'} + 2\log m + k + 1\\ \le 2\log m + 1 - k.\end{gathered}$$

If $k > m'/2$ then the second term comes into play. The worst-case upper bound on $\Delta\Phi$ occurs when π_1, the semipath containing the splitting vertex, has size k. $\Delta\Phi$ is at most

$$\begin{gathered}2(m'-k+1)\log\frac{m'-k+1}{m'} + 2k\log\frac{k}{m'} + 2\log m\\ + 1 + (m'-k) - (2k - m' - 1)\\ \le 2\log m - 2(m'-k) - 3k + 2m' = 2\log m - k.\end{gathered}$$

In both cases the amortized cost is $O(\log m)$. This completes the proof of the theorem. □

THEOREM 6.3. *Given a collection of path hulls and singleton vertices that contain a total of* m *vertices, we can perform an intermixed sequence of* $O(m)$ *tangent computations, vertex additions, vertex deletions, and path merges in* $O(m\log m)$ *total time.*

Proof. The idea of this proof is the same as that of the previous proof: we specify the costs of the operations precisely, define a potential function bounded by 0 and $2m\log m + m$, and show that the amortized cost of each operation is $O(\log m)$. As in the previous proof, the operation costs are $\log m'$ for tangent finding, $\log(d+2)$ for a vertex addition, and either 1 or m' for vertex deletions, depending on whether path hulls must be rebuilt. The cost of merging two paths to produce a single path of length m' is $k\log(m'/k)$, where k is the length of the shorter of the two paths.

We define the potential of a single path hull that has two portions of lengths m_1 and m_2, with $m_1 + m_2 = m' + 1$, to be

$$2m'\log(m/m') + |m_1 - m_2|.$$

The potential for the whole collection of path hulls, which we call Φ, is the sum of the individual potentials. It satisfies $0 \leq \Phi \leq 2m\log m + m$.

As in Theorem 6.2, the amortized cost of a tangent computation is the same as its true cost, $\log m' = O(\log m)$. Similarly, the amortized costs of vertex additions and deletions are $O(\log(m/m')) = O(\log m)$.

For a merge, $\Delta\Phi$ is at most

$$2(m' - k + 1)\log\frac{m' - k + 1}{m'} + 2k\log\frac{k}{m'} - 2\log\frac{m}{m'} + k - 1$$
$$\leq -2\log\frac{m}{m'} + 2k\log\frac{k}{m'} + k \leq -k\log\frac{m'}{k}.$$

Hence the amortized cost of a merge is $O(1)$. □

6.4. Storage allocation for semipath hulls. In this section we discuss how to allocate storage for the semipath hulls used in the algorithms of §§3 and 4. In order to represent the convex hull of a simple path by a semipath hull, we must allocate a block of storage proportional in size to the length of the path. During the rotation algorithm, the paths lengthen and shorten due to vertex additions and deletions. In the face of changing path lengths, we must ensure that each semipath hull has a block of storage whose minimum size is proportional to the length of the path. We describe three solutions to the allocation problem; these solutions trade conceptual complexity for space complexity.

The first solution is trivial: simply allocate $O(n)$ storage for each semipath hull. Each semipath hull has enough space; most have an excess. This scheme takes $O(n^2)$ storage.

The second solution is less trivial: copy semipath hulls when they get too big or too small. This is the approach used when $k > 1$, that is, when there are islands inside the outer polygon (§4). Each semipath hull $SPH(\pi, x)$ is initially given a block of storage appropriate for a path at most two times longer or shorter than π. Suppose that the block is appropriate for a path of length m. When the length of π becomes less than $m/4$ or greater than m, we copy the semipath hull into a block of storage half or twice as big, as appropriate. The copying cost is proportional to the number of vertex additions and deletions performed on the semipath hull since the last copying operation, and so the copying does not increase the asymptotic complexity of the semipath hull operations. A straightforward scheme for block allocation uses $O(kn\log n)$ space: for each i between 0 and $\log n$, we allocate $O(kn)$ storage in blocks of size $O(2^i)$. Because there are only $O(kn)$ path vertices total, and no path is larger than $O(n)$, there are always enough blocks of the required sizes.

The third solution reduces the storage to $O(n)$ in the simple polygon case ($k = 1$). This is the trickiest solution of the three: allocate three linear-size arrays to hold all the semipath hull data, then assign disjoint subarrays to the different semipath hull data blocks. Making sure the blocks stay disjoint as the semipath hulls change is the key to this approach.

The description of this solution is divided into three parts: keeping semipath hulls stationary, sharing a deque between two arrays, and indexing the semipath hull storage.

Section 6.1 shows that a semipath hull changes very little when a vertex is added or deleted. No vertices are moved except the one being added or deleted. Thus a requirement of a storage allocation scheme is that if π' is a subpath of π with the same anchor x, then the storage for $SPH(\pi', x)$ should be a contiguous subset of that for $SPH(\pi, x)$. Because the paths we deal with are subsequences of the polygon boundary, this suggests that we number the vertices around the boundary of the polygon and assign storage based on indices. A path with endpoints numbered i and j along the boundary of P would be assigned storage with indices $i..j$. This is the idea we use, although the indexing is not quite so simple.

Before we discuss indexing, we consider storing a deque in an array fixed at one end. The deque part of $SPH(\pi, x)$ grows at both ends as new vertices are added to π, but the array locations reserved for it extend in only one direction: the location assigned to vertex x is fixed. To allow double-ended growth, we use two arrays, *right* and *left*. We split the deque array described in §6 into a left half and a right half. If locations $x..v$ are reserved for the deque, then we store the right half of the deque in *right*$[x..v]$, growing toward v, and the left half in *left*$[x..v]$, also growing toward v. This complicates the deque indexing, but has the properties we need. The transcript stack is easy to manage; we just store it in a transcript array in locations indexed by $x..v$, with the stack growing toward v.

The array indexing scheme we use is not vertex-based, but edge-based. In the algorithm of §3, we maintain a semipath hull for each free-space edge on the boundary of B_α. If each of the associated paths lay outside B_α, in the bay cut off by the free-space edge, then assigning storage based on polygon vertex indices would work—path vertices would lie in disjoint intervals of the polygon boundary. However, the path associated with a free-space edge doesn't always lie in the bay cut off by the edge, for example when a leading free-space edge is retreating (moving to the right). To cope with this difficulty, we allocate the storage for a semipath hull based on the polygon and triangulation edges that cut the free-space edge. We assign indices 1 through $4n - 6$ to the endpoints of all polygon and triangulation edges, numbered consecutively along the polygon boundary. Semipath hulls are stored in three arrays with these indices. The path associated with a free-space edge f has as many vertices as there are polygon or triangulation edges that touch f from a particular side of the path. This number is no larger than the number of edge endpoints inside the bay that f cuts off, and so we can allocate the storage for the semipath hull from that associated with these edge endpoints. In particular, the triangulation or polygon edge from f to the anchor of the path determines the fixed end of the semipath hull storage.

7. An open problem.

In the standard model of compliant motion [2, 5, 8, 9, 11, 25], a vertex of the environment can be "sticky" when the robot encounters it coming from free space. In reality, however, if the robot "shakes" a little, it may get unstuck and reach the goal. In other words, it is possible that this vertex separates two good ranges for the starting point of the robot. Under the model in which the robot is able to "shake loose" when it gets stuck, it may be possible to unite these two ranges. This can produce a good range for the imperfect control case, even when no good range exists under the standard model. Figure 15 illustrates the problem: the presence of island I prevents the merging of the two good ranges for p, but if island I

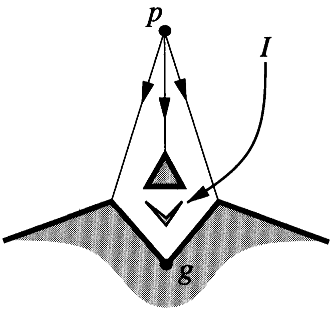


FIG. 15. *Should the triangle peak be sticky?*

were not there, merging would be possible. We conjecture that it is possible to merge ranges, given the control uncertainty of the robot, in $O(kn \log n)$ preprocessing and $O(k \log n)$ query time.

Acknowledgments. We would like to thank Jean-Claude Latombe for introducing us to the problem of compliant motion planning, to Leo Guibas for reading an early draft of this paper and for making important comments about Theorem 2.7, and to the referees for their comments.

REFERENCES

[1] M. BRADY, J. M. HOLLERBACH, T. L. JOHNSON, T. LOZANO-PEREZ, AND M. MASON, *Robot Motion: Planning and Control*, MIT Press, Cambridge, MA, 1982.

[2] A. J. BRIGGS, *An efficient algorithm for one-step planar compliant motion planning with uncertainty*, Algorithmica, 8 (1992), pp. 195–208.

[3] S. J. BUCKLEY, *Planning and Teaching Compliant Motion Strategies*, Ph.D. thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 1987.

[4] J. F. CANNY, *A new algebraic method for robot motion planning and real geometry*, in Proc. 28th IEEE Symposium on Foundations of Computer Science, IEEE Press, Piscataway, NJ, 1987, pp. 39–48.

[5] J. F. CANNY AND J. REIF, *New lower bound techniques for robot motion planning problems*, in Proc. 28th IEEE Symposium on Foundations of Computer Science, IEEE Press, Piscataway, NJ, 1987, pp. 49–60.

[6] B. CHAZELLE, *Triangulating a simple polygon in linear time*, Discrete Comput. Geom., 6 (1991), pp. 485–524.

[7] D. DOBKIN, L. GUIBAS, J. HERSHBERGER, AND J. SNOEYINK, *An efficient algorithm for finding the CSG representation of a simple polygon*, Algorithmica, 10 (1993), pp. 1–23.

[8] B. R. DONALD, *Error Detection and Recovery in Robotics*, Lecture Notes in Comput. Sci. 336, Springer-Verlag, New York, 1989.

[9] ———, *The complexity of planar compliant motion under uncertainty*, Algorithmica, 5 (1990), pp. 353–382.

[10] H. EDELSBRUNNER, L. J. GUIBAS, AND J. STOLFI, *Optimal point location in a monotone subdivision*, SIAM J. Comput., 15 (1986), pp. 317–340.

[11] M. A. ERDMANN, *Using backprojections for fine motion planning with uncertainty*, Internat. J. Robotics Res., 5 (1986), pp. 19–45.

[12] M. A. ERDMANN AND M. MASON, *An exploration of sensorless manipulation*, in Proc. IEEE International Conference on Robotics, IEEE Press, Piscataway, NJ, 1986, pp. 1569–1574.

[13] S. FORTUNE AND G. WILFONG, *Planning constrained motion*, in Proc. 20th ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1988, pp. 445–459.

[14] A. FOURNIER AND D. Y. MONTUNO, *Triangulating simple polygons and equivalent problems*, ACM Transactions on Graphics, 3 (1984), pp. 153–174.

[15] J. Friedman, J. Hershberger, and J. Snoeyink, *Compliant motion in a simple polygon*, in Proc. 5th ACM Symposium on Computational Geometry, Association for Computing Machinery, 1989, pp. 175–186.

[16] ———, *Input-sensitive compliant motion in the plane*, in Proc. 2nd Scandinavian Workshop on Algorithm Theory, Springer-Verlag, Berlin, 1990, pp. 225–237.

[17] M. R. Garey, D. S. Johnson, F. P. Preparata, and R. E. Tarjan, *Triangulating a simple polygon*, Inform. Process. Lett., 7 (1978), pp. 175–179.

[18] R. L. Graham and F. F. Yao, *Finding the convex hull of a simple polygon*, J. Algorithms, 4 (1983), pp. 324–331.

[19] L. J. Guibas, L. Ramshaw, and J. Stolfi, *A kinetic framework for computational geometry*, in Proc. 24th IEEE Symposium on Foundations of Computer Science, IEEE Press, Piscataway, NJ, 1983, pp. 100–111.

[20] L. J. Guibas, M. Sharir, and S. Sifrony, *On the general motion-planning problem with two degrees of freedom*, Discrete Comput. Geom., 4 (1989), pp. 491–521.

[21] P. J. Heffernan and J. S. B. Mitchell, *Structured visibility profiles with applications to problems in simple polygons*, in Proc. 6th ACM Symposium on Computational Geometry, Association for Computing Machinery, 1990, pp. 53–62.

[22] J. Hershberger and J. Snoeyink, *Computing minimum length paths of a given homotopy class*, in Proc. 2nd Workshop on Algorithms and Data Structures, Lecture Notes in Comput. Sci. 519, Springer-Verlag, New York, 1991, pp. 331–342.

[23] S. Huddleston and K. Mehlhorn, *A new data structure for representing sorted lists*, Acta Inform., 17 (1982), pp. 157–184.

[24] D. Kirkpatrick, *Optimal search in planar subdivisions*, SIAM J. Comput., 12 (1983), pp. 28–35.

[25] J.-C. Latombe, *Motion planning with uncertainty: The preimage backchaining approach*, Tech. Report STAN-CS-88-1196, Department of Computer Science, Stanford University, Stanford, CA, 1988.

[26] ———, *Robot Motion Planning*, Kluwer International Series in Engineering and Computer Science, SECS 0124, Kluwer Academic Publishers, Norwell, MA, 1991.

[27] T. Lozano-Pérez, M. T. Mason, and R. H. Taylor, *Automatic synthesis of fine-motion strategies for robots*, Internat. J. Robotics Res., 3 (1984), pp. 3–24.

[28] T. Lozano-Pérez and M. A. Wesley, *An algorithm for planning collision-free paths among polyhedral obstacles*, Comm. Assoc. Comput. Mach., 22 (1979), pp. 560–570.

[29] A. Melkman, *On-line construction of the convex hull of a simple polyline*, Inform. Process. Lett., 25 (1987), pp. 11–12.

[30] B. K. Natarajan, *On Moving and Orienting Objects*, Ph.D. thesis, Department of Computer Science, Cornell University, Ithaca, NY, 1986.

[31] C. Ó'Dúnlaing and C. K. Yap, *A "retraction" method for planning the motion of a disc*, J. Algorithms, 6 (1986), pp. 104–111.

[32] J. O'Rourke, *Computational Geometry in C*, Cambridge University Press, London, 1994.

[33] F. P. Preparata and M. I. Shamos, *Computational Geometry: An Introduction*, Springer-Verlag, New York, 1985.

[34] J. T. Schwartz, M. Sharir, and J. Hopcroft, eds., *Planning, Geometry, and Complexity of Robot Motion*, Ablex Series in Artificial Intelligence, Ablex, Norwood, NJ, 1987.

[35] M. Sharir and A. Schorr, *On shortest paths in polyhedral spaces*, SIAM J. Comput., 15 (1986), pp. 193–215.

[36] G. Toussaint and D. Avis, *On a convex hull algorithm for polygons and its applications to triangulation problems*, Pattern Recognition, 15 (1982), pp. 23–29.

SIAM J. COMPUT.
Vol. 25, No. 3, pp. 600–625, June 1996

ON-LINE ALGORITHMS FOR PATH SELECTION IN A NONBLOCKING NETWORK*

SANJEEV ARORA[†], F. T. LEIGHTON[‡], AND BRUCE M. MAGGS[§]

Abstract. This paper presents the first optimal-time algorithms for path selection in an optimal-size nonblocking network. In particular, we describe an N-input, N-output, nonblocking network with $O(N \log N)$ bounded-degree nodes, and an algorithm that can satisfy any request for a connection or disconnection between an input and an output in $O(\log N)$ bit steps, even if many requests are made at once. Viewed in a telephone switching context, the algorithm can put through any set of calls among N parties in $O(\log N)$ bit steps, even if many calls are placed simultaneously. Parties can hang up and call again whenever they like; every call is still put through $O(\log N)$ bit steps after being placed. Viewed in a distributed memory machine context, our algorithm allows any processor to access any idle block of memory within $O(\log N)$ bit steps, no matter what other connections have been made previously or are being made simultaneously.

Key words. nonblocking network, multibutterfly network, multi-Beneš network, routing algorithm

AMS subject classifications. 68M10, 68Q22, 90B12, 94C10

1. Introduction.

1.1. Definitions.

Nonblocking networks arise in a variety of communications contexts. Common examples include telephone systems and network architectures for parallel computers. In a typical application, there are $2N$ *terminals* (usually thought of as N *inputs* and N *outputs*) interconnected by switches that can be set to link the inputs to the outputs with node-disjoint paths according to a specified permutation. (Switches are also called nodes.) In a *nonblocking* network, the terminals and nodes are interconnected in such a way that any unused input–output pair can be connected by a path through unused nodes, no matter what other paths exist at the time. The 6-terminal graph shown in Figure 1.1, with inputs Bob, Ted, and Pat and outputs Vanna, Carol, and Alice, for example, is nonblocking because no matter which input–output pairs are connected by paths, there is a node-disjoint path linking any unused input–output pair. In particular, if Bob is talking to Alice and Ted is talking to Carol, then Pat can still call Vanna.

The notion of a nonblocking network has several variations. The nonblocking network in Figure 1.1 is an example of the most commonly studied type. This network is called a *strict-sense nonblocking connector*, because no matter what paths are established in the network, it is possible to establish a path from any unused input to any unused output. A slightly weaker notion is that of a *wide-sense nonblocking*

[*] Received by the editors October 2, 1991; accepted for publication (in revised form) October 26, 1994. This research was conducted while the first and third authors were affiliated with the Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139. This research was supported in part by Defense Advanced Research Projects Agency contracts N00014-87-K-825 and N00014-89-J-1988, Air Force contract AFOSR-89-0271, and Army contract DAAL-03-86-K-0171. A preliminary version of this paper appeared in *Proc. 22nd Annual ACM Symposium on Theory of Computing,* 1990, pp. 149–158.

[†] Department of Computer Science, Princeton University, Princeton, NJ 08544 (arora@cs.princeton.edu).

[‡] Mathematics Department and Laboratory for Computer Science, Massachusetts Institute of Technology, Cambrige, MA 02139 (ftl@math.mit.edu).

[§] School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213 (bmm@cs.cmu.edu).

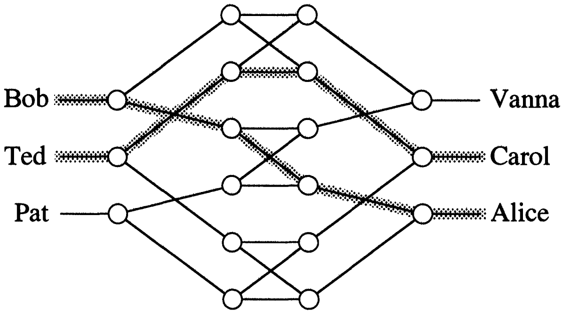


FIG. 1.1. *A nonblocking network with three inputs and three outputs.*

connector. A wide-sense nonblocking connector does not make the same guarantee as a strict-sense nonblocking connector. A network is a wide-sense nonblocking connector if there is an algorithm for establishing paths in the network, one after another, so that after each path is established, it is still possible to connect any unused input to any unused output. Still weaker is the notion of a *rearrangeable connector.* A rearrangeable connector is capable of realizing any one–one connection of inputs to outputs with node-disjoint paths provided that all the connections to be made are known in advance. A nonblocking or rearrangeable connector is a called a *generalized connector* if it has the additional property that each input can be simultaneously connected to an arbitrary set of outputs, provided that every output is connected to just one input. Generalized connectors are useful for multiparty calling in a telephone network as well as for broadcasting in a parallel machine.

1.2. Previous work. Nonblocking and rearrangeable networks have a rich and lengthy history. See [30] for an excellent survey and [9, 10] for more comprehensive descriptions of previous results. In 1950, Shannon [35] proved that any rearrangeable or nonblocking connector with N-inputs and N-outputs must have $\Omega(N \log N)$ edges.[1] Further work on lower bounds can be found in [4, 11, 32, 33]. In 1953, Clos constructed a strict-sense nonblocking connector with $O(N^{1+1/j})$ edges and depth j, for fixed j. (The degree of the nodes is not bounded.) Bounded-depth nonblocking networks have subsequently been studied extensively [8, 10, 24, 25, 29, 33]. In the early 1960s, Beizer [5] and Beneš [6] independently discovered bounded-degree rearrangeable connectors with depth $O(\log N)$ and size $O(N \log N)$, and Waksman [38] gave an elegant algorithm for determining how the nodes should be set in order to realize any particular permutation. Ofman [26] followed with a generalized rearrangeable connector of size $O(N \log N)$. Next, Cantor [7] discovered a bounded-degree $O(\log N)$-depth strict-sense nonblocking connector with $O(N \log^2 N)$ edges. The existence of a bounded-degree strict-sense nonblocking connector with size $O(N \log N)$ and depth $O(\log N)$ was first proved by Bassalygo and Pinsker [3]. Although the Bassalygo and Pinsker proof is not constructive, subsequent work on the explicit construction of expanders [23] yielded a construction.

More recent work has focused on the construction of generalized nonblocking connectors. In 1973, Pippenger [28] constructed a wide-sense generalized nonblocking connector with $O(N \log^2 N)$ edges. This result was later improved to $O(N \log N)$ edges by Feldman, Friedman, and Pippenger [10]. Recently, Turner suggested cascading two of the asymptotically larger Clos or Cantor networks as a more practical way

[1] Throughout this paper $\log N$ denotes $\log_2 N$.

to construct a generalized nonblocking connector [36]. This method requires that all the parties in a multiparty call are known at the time that the call is placed.

Unfortunately, there has not been as much progress on the problem of setting the nodes to realize the connection paths. Indeed, several of the references cited previously show that there exists a way of setting the nodes to realize the desired paths, but are unable to provide any reasonable algorithms for actually finding the right node settings. For example, no polynomial-time algorithm is known for finding the paths in the wide-sense generalized nonblocking connector of [10]. There are a few exceptions. On the naive nonblocking networks of size $\Theta(N^2)$ (e.g., an $N \times N$ mesh of trees [15]), a simple greedy algorithm suffices to find the paths on-line in $O(\log N)$ time. (An algorithm that finds the settings for the nodes is called a *circuit-switching* algorithm. An algorithm that is performed by the nodes themselves using only local information is called an *on-line* algorithm; an *off-line* algorithm is one that uses more global information.) Also, Lin and Pippenger recently found polylogarithmic-time off-line parallel algorithms for path selection in $O(N \log^2 N)$-size strict-sense nonblocking connectors using one processor per request [22]. On any strict-sense nonblocking connector, an on-line version of breadth-first search can be used to find a path from an unused input to an unused output on-line. Unfortunately, this algorithm cannot efficiently cope with simultaneous requests for connections. Nevertheless, no better algorithm, either on-line or off-line, was previously known for any $O(N \log N)$-size nonblocking network.

1.3. Models and conventions. The running times of the algorithms in this paper are described in two models, the *bit model* and the *word model.* In the *bit model*, each network node can be thought of as a finite automaton. In each *bit step*, the node can receive a single bit of information along each of its incoming edges (of which there are at most a constant number), change to a new state, and output a single bit of information on each of its outgoing edges (of which there are at most a constant number). In the *word model*, each edge in an N-node network can transmit a word consisting of up to $O(\log N)$ bits in a single step.

To simplify the explanation of the algorithms and results in this paper, we have adopted some conventions that may differ from the way that this material is treated in the more applied literature. For example, we generally route paths in a node-disjoint fashion. In practice, however, it may be desirable to route paths in an edge-disjoint manner instead. Our definitions and results can also be applied in this setting, as demonstrated in §5.3.3. Note that node-disjoint paths are automatically edge-disjoint, and any algorithm for routing edge-disjoint paths on a degree-d network can be converted into one for routing node-disjoint paths by replacing each node with a $d \times d$ complete bipartite graph.

1.4. Our results. In this paper, we describe an $O(N \log N)$-node nonblocking network for which each connection can be made on-line in $O(\log N)$ bit steps. The path selection algorithm works even if many calls are made at once—every call still gets through in $O(\log N)$ bit steps, no matter what calls were made previously and no matter what calls are currently active, provided that no two inputs try to access the same output at the same time. (If many inputs inadvertently try to access the same output at the same time, all but one of the inputs will receive a busy signal. The busy signals are also returned in $O(\log N)$ bit steps, but, at present, we require the use of a sorting circuit [2, 20] to generate the busy signals. Alternatively, we could merge the calling parties together, but this also requires the use of a sorting circuit.) In all scenarios, the size of the network and the speed of the path-selection algorithm are asymptotically optimal.

In addition to providing the first optimal solution to the abstract telephone-switching problem, our results significantly improve upon previously known algorithms for bit-serial packet routing. Previously, $O(\log N)$-bit-step algorithms for packet routing were known only for the special case in which all packet paths are created or destroyed at the same time, and even then only by resorting to the Ajtai–Komlós–Szemerédi (AKS) sorting circuit ([2]), or by using randomness on the hypercube [1]. In many circuit-switched parallel machines, however, packets are of varying lengths and packet paths are created and destroyed at arbitrary times, thereby requiring that paths be routed in a nonblocking fashion—something that previously discovered algorithms were not capable of doing. Even without worrying about the nonblocking property, our results provide the first non-AKS $O(\log N)$-bit-step algorithms for bit-serial packet routing on a bounded-degree network. (Since this work first appeared, Leighton and Plaxton have developed an $O(\log N)$-bit-step randomized sorting algorithm for the butterfly [20].)

1.5. Our approach. The networks that we use to obtain these results are constructed by combining expanders and Beneš networks in much the same way that expanders and butterflies are combined to form the multibutterfly networks described by Upfal [37]. We refer to these networks as *multi-Beneš networks.* The nonblocking networks of Bassalygo and Pinsker [3] are similar. The details of the construction are provided in §2 of the paper.

The techniques in this paper can also be applied to bandwidth-limited switching networks such as fat-trees [21]. These networks may be more useful in the context of real telephone systems, where there are limitations on the number of calls based on the proximity of the calls (e.g., it is unlikely that everyone on the East Coast will call everyone on the West Coast at the same time).

The description and analysis of the path-selection algorithm is divided into three sections. In §3, we prove that the multi-Beneš network is a strict-sense nonblocking connector. A similar approach was used in [17] to show that the multibutterfly is capable of routing in the presence of many faulty nodes. Indeed, we can think of currently used nodes as being faulty since they cannot be used to form new connections. Similarly, the algorithms we describe for routing in nonblocking networks can easily be extended to be highly tolerant of faults in the network. In §4, we describe an $O(\log N)$-bit-step algorithm for bit-serial routing in a multibutterfly. This algorithm relies on an unshared-neighbor property possessed by all highly expanding graphs. By implementing this algorithm on the multi-Beneš network and combining it with the methods of §3, we produce an algorithm that can handle many calls at the same time, independent of what calls have been made previously and what calls are currently connected.

In §5, we describe algorithms for handling multiparty calls, and situations where many inputs try to reach the same output simultaneously. Some of these algorithms rely on sorting circuits and are not as practical as those described in §4. We also show how to remove the distinction between terminals and nonterminals.

2. The multi-Beneš and multibutterfly networks.

Our nonblocking network is constructed from a Beneš network in much the same way that a multibutterfly network [37] is constructed from a butterfly network. We start by describing the butterfly, Beneš, and multibutterfly networks.

An N-input butterfly has $\log N + 1$ levels, each with N-nodes. An example is shown in Figure 2.1. The Beneš network is a $(2\log N + 1)$-level network consisting

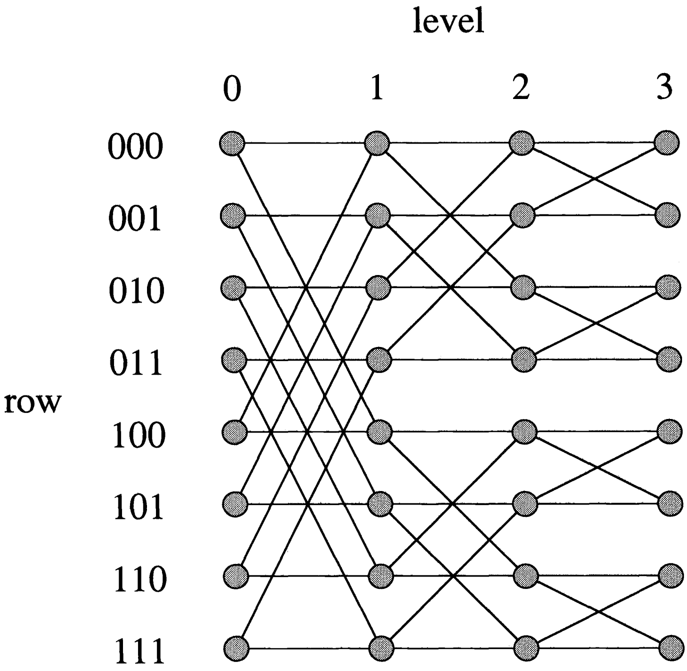


FIG. 2.1. *An eight-input butterfly network.*

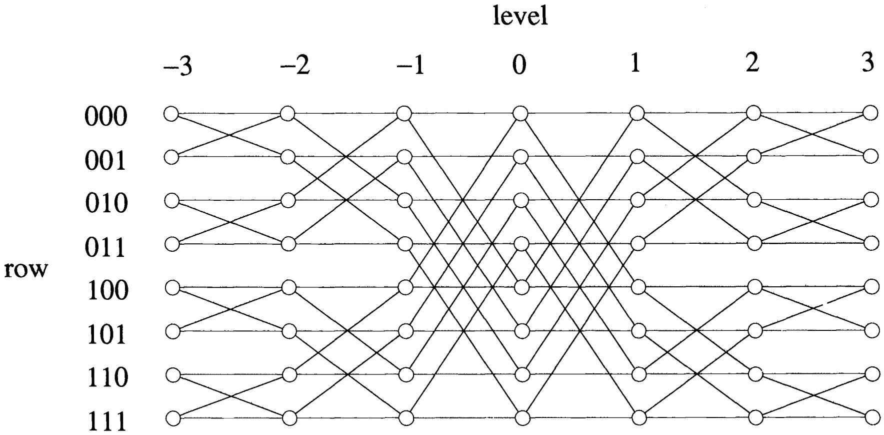


FIG. 2.2. *An eight-input Beneš network.*

of back-to-back butterflies. The network in Figure 2.2 is a Beneš network. Although Beneš networks are usually drawn with the long diagonal edges at the first and last levels rather than in the middle (see, e.g., [16, Fig. 3-27]), the networks are isomorphic.

A multibutterfly is formed by gluing together butterflies in a somewhat unusual way. In particular, given two N-input butterflies G_1 and G_2 and a collection of permutations $\Pi = \langle \pi_0, \pi_1, \ldots, \pi_{\log N} \rangle$, where $\pi_l : [0, \frac{N}{2^l} - 1] \rightarrow [0, \frac{N}{2^l} - 1]$, a two-butterfly is formed by merging the node in row $\frac{iN}{2^l} + i$ of level l of G_1 with the node

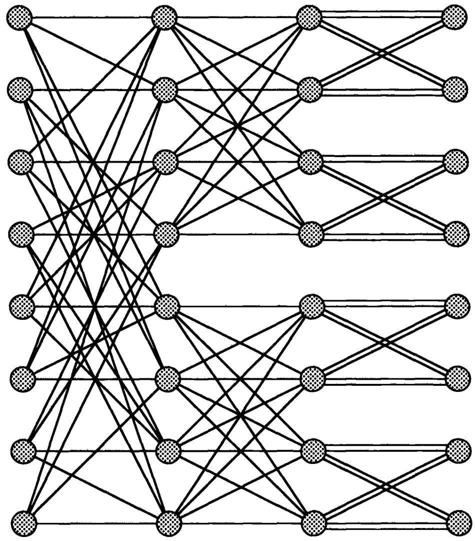


FIG. 2.3. *An eight-input two-butterfly network.*

in row $\frac{jN}{2^l} + \pi_l(i)$ of level l of G_2 for all $0 \le i \le \frac{N}{2^l} - 1$, all $0 \le j \le 2^l - 1$, and all $0 \le l \le \log N$. The result is an N-input $(\log N + 1)$-level graph in which each node has four inputs and four outputs. Of the four output edges at a node, two are *up* outputs and two are *down* outputs (with one up edge and one down edge coming from each butterfly). For an example, see Figure 2.3. Multibutterflies (i.e., d-butterflies) are composed from d butterflies in a similar fashion using $d-1$ sets of permutations, $\Pi^{(1)}, \ldots, \Pi^{(d-1)}$, where $\Pi^{(i)} = \{\pi_l^{(i)}, 0 \le l \le \log N\}$, resulting in a $(\log N + 1)$-level network with $2d \times 2d$ nodes.

In a butterfly or multibutterfly, for each output v there is a distinct *logical* (up-down) path from the inputs to v. In order to reach v from any input u, the path from u to v must take an up-edge from level l to level $l+1$ if the lth bit in the row number of v is 0, and a down-edge if the bit is 1. (The bits are counted starting with the most significant, which is in position 0.) Figure 2.4 shows the logical path from any input to output 011. Let us use the term *physical path* to denote our usual notion of a path through the network, i.e., a physical path consists of a sequence of nodes $w_0, w_1, \ldots, w_{\log N}$ such that node w_i resides on level i of the network, and nodes w_i and w_{i+1} are connected by an edge, for $0 \le i < \log N$. In a butterfly network, the logical path can be realized by only one physical path through the network. In a multibutterfly, however, each step of the logical path can be taken on any one of d edges. Hence, for any logical path there are many physical paths through the network.

The notion of up and down edges can be formalized in terms of splitters. More precisely, the edges from level l to level $l+1$ in rows $\frac{jN}{2^l}$ to $\frac{(j+1)N}{2^l} - 1$ in a multibutterfly form a *splitter* for all $0 \le l < \log N$ and $0 \le j \le 2^l - 1$. Each of the 2^l splitters starting at level l has $\frac{N}{2^l}$ inputs and $\frac{N}{2^l}$ outputs. The outputs on level $l+1$ are naturally divided into $\frac{N}{2^{(l+1)}}$ *up* outputs and $\frac{N}{2^{(l+1)}}$ *down* outputs. By definition, all splitters on the same level l are isomorphic, and each input is connected to d up outputs and d down outputs according to the butterfly and the permutations $\pi_l^{(1)}, \ldots, \pi_l^{(d-1)}$ and $\pi_{l+1}^{(1)}, \ldots, \pi_{l+1}^{(d-1)}$.

The most important characteristic of a multibutterfly is the set of permutations

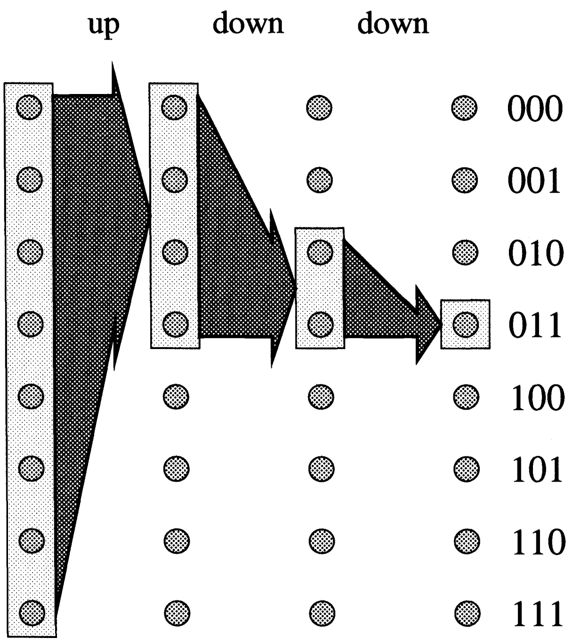


FIG. 2.4. *The logical up-down path from any input to output* 011.

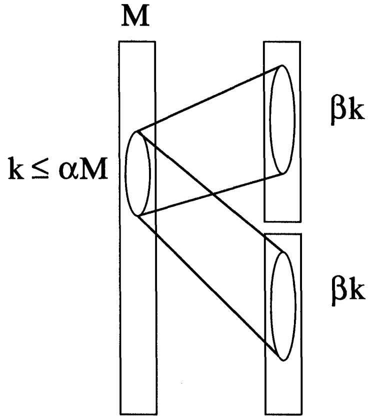


FIG. 2.5. *A splitter with expansion property* (α, β).

$\Pi^{(1)}, \ldots, \Pi^{(d-1)}$ that prescribe the way in which the component butterflies are to be merged. For example, if all the permutations are the identity map, then the result is the *dilated butterfly* (i.e., a butterfly with d copies of each edge). We are most interested in multibutterflies that have expansion properties. In particular, we say that an M-input splitter has *expansion property* (α, β) if every set of $k \leq \alpha M$ inputs is connected to at least βk up outputs and βk down outputs for $\beta > 1$. Similarly, we say that a multibutterfly has *expansion property* (α, β) if each of its component splitters has expansion property (α, β). For example, see Figure 2.5.

Although the constants α, β, and d do not appear in the expressions for the

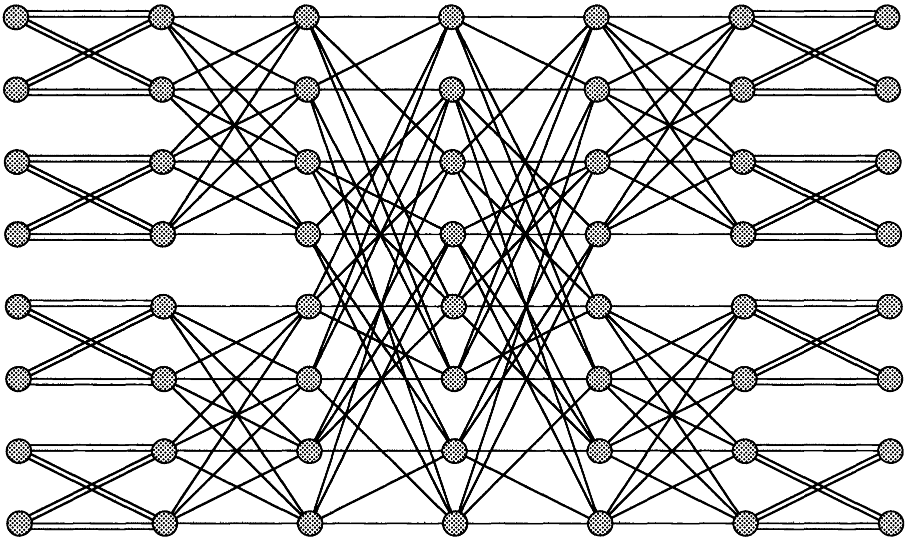


FIG. 2.6. *An eight-input two-multi-Beneš network.*

running times of our algorithms, e.g., $O(\log N)$, as a practical matter they are crucial. In general, the larger β is, the fewer bit steps an algorithm will require. However, since $d \geq \beta$, a network with large β must also have large d, and in practice it may be difficult to build a node that can receive and transmit along all d of its edges simultaneously if d is large. Furthermore, most of the algorithms require $\beta > d/2$, which (as far as we know) can only be achieved for small α. As we shall see, the fraction of network nodes that are actually used by paths is at most $1/\alpha$, so if α is small, the network is not fully utilized.

If the permutations $\Pi^{(1)}, \ldots, \Pi^{(d-1)}$ are chosen randomly, then with nonzero probability, the resulting d-butterfly has expansion property (α, β) for any d, α, and β for which $2\alpha\beta < 1$ and

$$d > \beta + 1 + \frac{\beta + 1 + \ln 2\beta}{\ln(\frac{1}{2\alpha\beta})}. \tag{2.1}$$

This bound appears as Corollary 2.1 in [37]. A derivation can be found in [18]. Roughly speaking, the bound says that the expansion, β, can be almost as large as $d - 1$, provided that α is small enough. Furthermore, for any α, β can be made arbitrarily close to $1/2\alpha$, by making d large. It is not known if β can be made close to both $d - 1$ and $1/2\alpha$ simultaneously. Constructions for splitters and multibutterflies with good expansion properties are known, although the expansion properties are generally not as good as those obtained from randomly generated graphs.

Like a multibutterfly, a multi-Beneš network is formed from Beneš networks by merging them together. A 2-multi-Beneš network is shown in Figure 2.6. An N-input multi-Beneš network has $2\log N + 1$ levels labeled $-\log N$ through $\log N$. Levels 0 through $\log N$ form a multibutterfly, while levels $-\log N$ through 0 form the mirror image of a multibutterfly.

As in the multibutterfly, the edges in levels 0 through $\log N$ are partitioned into splitters. Between levels $-\log N$ and 0, however, the edges are partitioned into *mergers*. More precisely, the edges from level l to level $l+1$ in rows $j2^{l+\log N+1}$ to $(j+1)2^{l+\log N+1}-1$ form a *merger* for all $-\log N \leq l < 0$ and $0 \leq j \leq N/2^{l+\log N+1}-1$. Each of the $N/2^{l+\log N+1}$ mergers starting at level l has $2^{l+\log N+1}$ inputs and outputs. The inputs on level l are naturally divided into $2^{l+\log N}$ *up* inputs and $2^{l+\log N}$ *down* inputs. All mergers on the same level l are isomorphic, and each input is connected to $2d$ outputs. There is a single, trivial, logical path from any input of a multi-Beneš network through the mergers on levels $-\log N$ through -1 to the single splitter on level 0. (Any physical path will do.) From level 0 there is a single logical up-down path through the splitters to any output on level $\log N$. In both cases, the logical path can be realized by many physical paths.

We say that an M-output merger has expansion property (α, β) if every set of $k \leq \alpha M$ inputs (up or down or any combination) is connected to at least $2\beta k$ outputs, $\beta > 1$. With nonzero probability, a random set of permutations yields a merger with expansion property (α, β) for any d, α, and β for which $\alpha\beta < 1/2$ and

$$2d > 2\beta + 1 + \frac{2\beta + 1 + \ln 2\beta}{\ln(\frac{1}{2\alpha\beta})}. \tag{2.2}$$

This inequality can be derived by making a small number of substitutions in the derivation of Inequality 2.1 found in [18]. We say that a multi-Beneš network has expansion property (α, β) if each of its component mergers and splitters has expansion property (α, β). The multibutterflies and multi-Beneš networks considered throughout this paper are assumed to have expansion property (α, β).

It is worth noting that all the results in this paper hold for a broader class of networks than multibutterflies and multi-Beneš networks. In particular, each basic butterfly component used to make a multibutterfly or multi-Beneš network can be replaced by any Delta network. A *Delta* network is a regular network formed by splitters like the butterfly, but for which the individual connections within each splitter can be arbitrary [14].

3. A proof that the multi-Beneš network is nonblocking. In this section we prove that the multi-Beneš network is a strict-sense nonblocking connector. As a consequence, a simple algorithm like breadth-first search can be used to establish a single path from any unused input to any unused output in $O(\log N)$ bit steps, where N is the number of rows. Algorithms that handle simultaneous requests for connections and multiparty calls are deferred to §§4 and 5.

In order for the algorithm to succeed, the multi-Beneš network must be "lightly loaded" by some fixed constant factor L, where we will choose L to be a power of 2. Thus, in an N-row multi-Beneš network, we only make connections between the N/L inputs and outputs in rows that are multiples of L. Since the other inputs and outputs are not used, the first and last $\log L$ levels of the network can be removed, and the N/L inputs and outputs can each be connected directly to their L descendants and ancestors on levels $-\log N + \log L$ and $\log N - \log L$, respectively.

The basic idea is to treat the nodes through which paths have already been established as if they were faulty and to apply the fault propagation techniques from [17] to the network. In particular, we define a node to be *busy* if there is a path currently routing through it. We recursively define a node in the second half of the network to be *blocked* if all of its up outputs or all of its down outputs are busy or blocked. More precisely, nodes are declared to be blocked according to the following rule. Working backwards from level $\log N - \log L - 1$ to level 0, a node is declared blocked if either all d of its up edges or all d of its down edges lead to busy or blocked nodes. From level -1 to level $-\log N + \log L$, a node is declared blocked if all $2d$ of its outgoing edges lead to busy or blocked nodes. A node that is neither busy nor blocked is said to be *working.*

The following pair of lemmas bound the fraction of input nodes that are blocked in every splitter and merger.

LEMMA 3.1. *For $L > 1/2\alpha(\beta - 1)$, at most a 2α fraction of the inputs in any splitter are declared to be blocked. Furthermore, at most an α fraction of the inputs are blocked because of busy and blocked nodes from the upper outputs, and at most an α fraction are blocked because of busy and blocked nodes from the lower outputs.*

Proof. The proof is by induction on level number, starting at level $\log N - \log L$ and working backwards to level 0. The base case is trivial since there are no blocked nodes on level $\log N - \log L$. Suppose the inputs of an M-input splitter contain more than αM nodes that are blocked because of the upper (say) outputs. Consider the set U of busy or blocked upper outputs. Since all of the edges out of a blocked input lead to busy or blocked outputs, we can conclude that $|U| \geq \alpha\beta M$. Since every path passing through the upper outputs must lead to one of $M/2L$ terminals, there can be at most $M/2L$ busy nodes among the upper outputs of the splitter. Furthermore, by induction there are at most αM blocked nodes among the upper outputs. Thus, $|U| \leq \alpha M + M/2L$. For $L > 1/2\alpha(\beta - 1)$ we have a contradiction. Hence, at most an α fraction of the nodes are blocked, as claimed. □

LEMMA 3.2. *For $L > 1/2\alpha(\beta - 1)$, at most a 2α fraction of the upper inputs and a 2α fraction of the lower inputs in any merger are blocked.*

Proof. The proof is like that of Lemma 3.1. □

After the fault-propagation process, every working node in the first half of the network has an output that leads to a working node, and every working node in the second half has both an up output and a down output that lead to working nodes. Furthermore, since at most a 2α fraction of the nodes in each merger on level $-\log N + \log L$ is blocked, and $2\alpha L < L - 1$ for $L > 1/2\alpha(\beta - 1)$ and $2\alpha\beta < 1$, each of the N/L inputs has an edge to a working node on level $-\log N + \log L$. As a consequence, we can establish a path through working nodes from any unused input to any unused output in $O(\log N)$ bit steps using a simple greedy algorithm. Since the declaration of blocked nodes takes just $O(\log N)$ bit steps, and since the greedy routing algorithm is easily accomplished in $O(\log N)$ bit steps, the entire process takes just $O(\log N)$ bit steps.

The preceding algorithm for establishing paths one after another in the multi-Beneš network implies that it is a wide-sense nonblocking connector. The proofs of Lemmas 3.1 and Lemmas 3.2, however, do not make any assumptions about the strategy used to make previous connections between inputs and outputs. Indeed, the only requirement is that there are at most M/L paths through each M-input splitter or M-output merger, which holds for any path-selection strategy. Therefore, no matter how the paths for the previous connections were found, there is still at least one working node in each block at level $-\log N + \log L$, and as a consequence, at least one path between any unused input and unused output. Thus the multi-Beneš network is also a strict-sense nonblocking connector. As such, it is not really necessary to label the nodes as blocked or working; a simple on-line algorithm-like breadth-first search is guaranteed to find a path. When simultaneous requests are dealt with in §4.4, however, proper labeling will be important.

4. Establishing many paths at once.

In this section, we describe an on-line algorithm for routing an arbitrary number of additional calls in $O(\log N)$ bit steps.

As before, we assume for the time being that each input and each output is involved in at most one two-party call. Extensions to the algorithm for handling multiparty calls are described in §5. We also assume that paths are established between inputs and outputs on rows congruent to 0 mod L in the multi-Beneš network, where L is a power of 2 and $L \geq 1/\alpha$. This will ensure that no splitter or merger is ever overloaded.

To simplify the exposition of the algorithm, we start by describing an on-line algorithm for routing any initial set of paths in a multibutterfly (i.e., we don't worry about the nonblocking aspect of the problem for the time being). This comprises the first known circuit-switching algorithm for the multibutterfly. (Previous routing algorithms for the multibutterfly [17, 37] only worked for the store-and-forward model of routing.) The existence of the circuit-switching algorithm provides another proof that the multibutterfly is a rearrangeable connector. We conclude by modifying the definitions of busy and blocked nodes from §3 and showing how to implement the circuit-switching algorithm on a multi-Beneš network so that it works even in the presence of previously established calls.

4.1. Unshared neighbors. Our circuit-switching algorithm requires the splitters in the multibutterfly to have a special "unshared-neighbors" property defined as follows.

DEFINITION 4.1. *An M-input splitter is said to have the (α, δ) unshared-neighbors property if in every subset X of $k \leq \alpha M$ inputs, there are δk nodes in X that have an up-output neighbor that is not adjacent to any other node in X, and there are δk nodes in X that have a down-output neighbor that is not adjacent to any other node in X (i.e., δk nodes in X have an unshared up-neighbor, and δk nodes have an unshared down-neighbor).*

LEMMA 4.2. *Any splitter with the (α, β) expansion property has the (α, δ) unshared-neighbors property, where $\delta = 2\beta/d - 1$, provided that $\beta > d/2$.*

Proof. Consider any set X of $k \leq \alpha M$ inputs in an M-input splitter. These nodes have at least βk neighbors among the up (down) outputs. Let n_1 denote the number of these neighbors adjacent to precisely one node of X, and let n_2 denote the number of neighbors adjacent to two or more nodes of X. Then $n_1 + n_2 \geq \beta k$ and $n_1 + 2n_2 \leq dk$. Solving for n_1 reveals that $n_1 \geq (2\beta - d)k$. Hence at least $(2\beta/d - 1)k$ of the nodes in X are adjacent to an unshared neighbor. □

By (2.1), we know that randomly generated splitters have the (α, δ) unshared-neighbors property, where δ approaches 1 as d gets large and α gets small. Explicit constructions of such splitters are not known, however. Nevertheless, we will consider only multibutterflies with the (α, δ) unshared-neighbors property for $\delta > 0$ in what follows.

Remark. The (α, β) expansion property ($\beta > d/2$) is a sufficient condition for the unshared-neighbors property, but by no means necessary. In fact, we can easily prove the existence of random splitters that have a fairly strong (α, δ) unshared-neighbors property for small degree. For such graphs, the routing algorithm we are about to describe is more efficient in terms of hardware required. However, multibutterflies with expansion properties will remain the object of our focus.

4.2. A level-by-level algorithm. Our first algorithm extends the paths from level 0 to level $\log N$ by first extending all the paths from level 0 to level 1, then from level 1 to level 2, and so on. As we shall see, extending the paths from one level to the next can be done in $O(\log N)$ bit steps, so the total time is $O(\log^2 N)$ bit steps.

In a multibutterfly with the (α, δ) unshared-neighbors property, it is relatively easy to extend paths from one level to the next because paths at nodes with unshared neighbors can be extended without worrying about blocking any other paths that are trying to reach the next level. The remaining paths can then be extended recursively. In particular, all the paths can be extended from level l to level $l+1$ (for any l), by performing a series of "steps," where each step consists of

1. every path that is waiting to be extended sends out a "proposal" to each of its output (level $l+1$) neighbors in the desired direction (up or down),
2. every output node that receives precisely one proposal sends back its acceptance to that proposal,
3. every path that receives an acceptance advances to one of its accepting outputs on level $l+1$.

Note that each step can be implemented in a constant number of bit steps.

Since the splitters connecting level l to level $l+1$ have $M = N/2^l$ inputs, and at most $M/2L$ paths must be extended to the upper (or lower) outputs, for $L > 2/\alpha$, the number of inputs containing these paths is at most αM. Thus, we can apply the (α, δ) unshared-neighbors property to these nodes. As a consequence, in each step the number of paths still remaining to be extended decreases by a $(1-\delta)$ factor. After $\log(N/L2^{l+1})/\log(1/(1-\delta))$ steps, no paths remain to be extended.

By using the path-extension algorithm just described to extend all of the paths from level 0 to level 1, then all of the paths from level 1 to level 2, and so on, we can construct all the paths in

$$\sum_{l=0}^{\log\frac{N}{L}-1} \frac{\log\frac{N}{L2^{l+1}}}{\log\frac{1}{1-\delta}} \le \frac{\log^2\frac{N}{2L}}{\log\frac{1}{1-\delta}} = O(\log^2 N)$$

steps.

4.3. A faster algorithm. To construct the paths in $O(\log N)$ bit steps we modify the first algorithm as follows. Given a set of at most αM paths that need to be extended at an M-input splitter, the algorithm does not wait $\Theta(\log M)$ time for every path to be extended before it begins the extension at the next level. Instead, it waits only $O(1)$ steps, in which time the number of unextended paths falls to a fraction ρ of its original value. We will choose ρ to be less than $1/d$. Now the path extension process can start at the next level. The only danger here is that the ρ fraction of paths left behind may find themselves blocked by the time they reach the next level, and so we need to ensure that this won't happen. Therefore, stalled paths send out *placeholders* to all of their neighbors at the next level, and henceforth the neighbors with placeholders participate in the path-extension process at the next level as if they were paths. Thus, a placeholder not only reserves a spot that may be used by a path at a future time, but also helps to chart out the path by continuing to extend ahead. Since a placeholder doesn't know which path will ultimately use it, a node holding a placeholder must extend paths into both the upper and lower output portions of its splitter. A placeholder that first extends a path into the upper output portion of its splitter continues to attempt to extend a path into the lower portion, and vice versa. We will call a path from an input of the network to an input of a splitter in the network a *real path* if it contains no placeholders. The goal of the algorithm, of course, is to extend real paths all the way through the network. Any path that contains at least one placeholder is called a *placeholder path.*

Since each stalled path generates up to $2d$ placeholders at the next level, and these placeholders might later become stalled themselves, there is a risk that the network will become clogged with placeholders. In particular, if the fraction of inputs in a splitter that are trying to extend rises above α, the path-extension algorithm ceases to work. Thus, in order to prevent placeholders from clogging the system, whenever a stalled path, either real or a placeholder, gets extended into either the upper or lower output portion of a splitter, it sends a *cancellation signal* to each of the nodes in that portion of the splitter that are holding placeholders for it. When a placeholder is replaced by a real path, one of the two directions (up or down) into which the placeholder has been attempting to extend becomes unnecessary. If the placeholder has already extended its path in that direction, a single cancellation is sent along the edge that the path uses. Otherwise, a cancellation is sent to each of the d placeholding neighbors in that direction. When a placeholding node gets cancellations from all of the nodes that had requested it to hold their places, it ceases its attempts to extend. It also sends cancellations to any nodes ahead of it that may be holding a place for it. Note that a placeholding node that has received cancellations from all but one of the nodes that had requested it to hold their places continues to try to extend into both the upper and lower output portions of the splitter. As we shall see, this scheme of cancellations prevents placeholders from getting too numerous.

The $O(\log N)$-step algorithm for routing paths proceeds in *phases*. Each path is restricted to extend forward by at most one level during each phase. We refer to the first wave of paths and placeholders to arrive at a level as the *wavefront*. The wavefront moves forward by one level during each phase. A phase consists of the following three parts:

(i) C steps of passing cancellation signals. These cancellation signals travel at the rate of one level per step.

(ii) T steps of extending paths from one level to the next. In this time, the number of stalled (i.e., unextended) paths at each splitter drops by least a factor of ρ, where $\rho \leq (1-\delta)^T$.

(iii) one step of sending placeholders to all neighbors of paths in the wavefront that were not extended during the preceding T steps.

Note that for constant T and C, each phase can be performed in $O(1)$ bit steps. We will assume that $C \geq 2$ so that cancellation signals have a chance to catch up with the wavefront, and that $d \geq 3$.

The key to our analysis of the algorithm is to focus on the number of stalled paths (corresponding to real paths *or* placeholders) at the inputs of each splitter. In phase t of the algorithm, where the first phase is phase 0, the wavefront advances from level t to level $t+1$. Let P_i denote the maximum fraction of inputs containing wavefront paths (real or placeholder) in a level i splitter that wish to extend to the upper (or, similarly, to the lower) outputs of a level $i+1$ splitter at the beginning of phase i, i.e., when the wavefront arrives at level i, and let $S(i,t)$ denote the maximum fraction of inputs that contain stalled paths that wish to extend to the upper (or, similarly, to the lower) outputs of any splitter at level i at the end of phase t. Note that $S(i,t)=0$ for $t<i$, since there are no paths to extend at level i before phase i. Also, note that $S(i,i) \leq \rho P_i$.

The following lemmas will be useful in proving that every path is extended to completion in $\log N$ phases provided that $L \geq 1/\alpha$ and $\rho < 1/14d$.

LEMMA 4.3. *If $P_i \leq \alpha$ then $S(i,t) \leq \rho^{t-i} S(i,i) \leq \rho^{t+1-i} P_i$ for $t \geq i$.*

Proof. In each phase of the algorithm, the number of stalled paths at the inputs drops by a factor of ρ, provided that the number of paths trying to extend is never greater than an α fraction of the inputs of the splitter. Since the number of paths reaching the inputs never increases after the wavefront arrives, this condition is always satisfied. □

The following lemma bounds the size of the wavefront in terms of the number of stalled paths behind it.

LEMMA 4.4.

$$P_i \leq \frac{1}{2L} + 2dS(i-1,i-1) + \sum_{l=0}^{\infty}\sum_{k=1}^{C} 2^{Cl+k+1} dS(i-1-Cl-k, i-l-2).$$

Proof. The first term, $1/2L$, is an upper bound on the fraction of inputs through which real paths that wish to extend to the upper outputs (or, similarly, to the lower outputs) will ever pass. The $2dS(i-1,i-1)$ term represents the fraction of inputs that could hold placeholders generated by stalled paths at level $i-1$ (the factor of 2 comes in because the number of inputs in a splitter at level $i-1$ is twice as many as those in a level i splitter). The $4dS(i-2,i-2)$ term ($l=0$, $k=1$) is an upper bound on the fraction of inputs containing placeholders that were generated by paths stalled at level $i-2$ when the wavefront was extended to level $i-1$ in phase $i-2$. Next, for $C \geq 2$, the contribution of placeholders from level $i-3$ is $8dS(i-3,i-2)$ (here $l=0$, $k=2$), not $8dS(i-3,i-3)$, since paths that are stalled at level $i-3$ during phase $i-3$, but get through during phase $i-2$, send cancellation signals to levels $i-2$ and $i-1$ during the first part of phase $i-1$. Hence, these paths do not contribute placeholders to the wavefront when it is extended from level $i-1$ to level i. The contribution from level $i-C-2$ is $2^{C+2}dS(i-C-2,i-3)$ (here $l=1$, $k=1$), since paths that are extended during the second part of phase $i-3$ send cancellations that reach level $i-2$ during the first part of phase $i-2$. These cancellations then reach level $i-1$ during the first part of phase $i-1$. The rest of the terms in the summation may be counted similarly. Although our summation seems to have infinitely many terms, only finitely many of them are nonzero. □

The next lemma, Lemma 4.5, presents a weaker bound on P_i. The difference between this lemma and the previous one is that in Lemma 4.5 we assume that a cancellation signal must reach level i rather than $i-1$ before the start of the path-extension part of phase $i-1$ in order for it to have an effect on the size of the wave propagating from level $i-1$ to level i. The reason for this assumption is that we will later speed up the algorithm by overlapping the cancellation-passing and path-extension parts of each phase.

LEMMA 4.5.

$$\begin{aligned} P_i \leq & \frac{1}{2L} + 2dS(i-1,i-1) \\ & + \sum_{k=2}^{C} 2^k dS(i-k,i-2) \\ & + \sum_{l=1}^{\infty}\sum_{k=1}^{C} 2^{Cl+k} dS(i-Cl-k, i-l-2). \end{aligned}$$

Proof. The proof is similar to that of Lemma 4.4. □

The following lemma shows that for the right choices of L, ρ, d, and C, no splitter ever receives too many paths (real or placeholder) that want to extend to the upper outputs (and, similarly, to the lower outputs).

LEMMA 4.6. *For $L \geq 1/\alpha$, $\rho \leq 1/14d$, $d \geq 3$, and $C \geq 3$, $P_i \leq \alpha$, for $0 \leq i \leq \log(N/L)$.*

Proof. We prove by induction on i that for $\gamma = \alpha/14d$, $P_i \leq \alpha$, and $S(i,i) \leq \rho P_i \leq \gamma$. For the base case, observe that $P_0 \leq 1/2L$, and $S(0,0) \leq \rho P_0$ (by applying Lemma 4.3 with $i = 0$ and $t = 0$). Hence, $S(0,0) \leq \alpha/28d = \gamma/2$. For the inductive step, we apply Lemma 4.3 to the recurrence of Lemma 4.5, which yields

$$\begin{aligned} P_i &\leq \frac{1}{2L} + 2d\gamma + \sum_{k=2}^{C} 2^k d\gamma\rho^{k-2} \\ &\quad + \sum_{l=1}^{\infty}\sum_{k=1}^{C} 2^{Cl+k} d\gamma\rho^{(C-1)l+k-2} \\ &= \frac{1}{2L} + 2d\gamma + \frac{4d\gamma(1-(2\rho)^{C-1})}{1-2\rho} \\ &\quad + \frac{d\gamma 2^{C+1}\rho^{C-2}(1-(2\rho)^C)}{(1-2^C\rho^{C-1})(1-2\rho)} \\ &\leq \frac{1}{2L} + 2d\gamma + 4.2d\gamma + .5d\gamma. \end{aligned}$$

Note that in the last inequality we have used the fact that $d \geq 3$, $C \geq 3$, and $\rho \leq 1/14d$. (We really only needed $C \geq 2$, but the constants are better for $C \geq 3$.) Thus if $\gamma = \alpha/14d$ and $L \geq 1/\alpha$, then $P_i \leq \alpha$. Also, by Lemma 4.3, $S(i,i) \leq \rho P_i$ and if $\rho \leq 1/14d$, we have $S(i,i) \leq \alpha/14d = \gamma$, thereby establishing the induction. □

From Lemma 4.6, it is clear that no splitter ever has more than an α fraction of its inputs containing paths to be extended to the upper (or lower) outputs. Therefore the path-extension algorithm is never swamped by placeholders and always works as planned at each level, cutting down the number of stalled paths by a factor of ρ during each phase. Hence, $\log(\alpha M)/\log(1/\rho)$ phases after the wavefront arrives at a splitter of size M, all paths are extended. Since the wavefront arrives at level i during phase $i-1$, the algorithm establishes all real paths to level $\log(N/L)$ (recall that the last $\log L$ levels have been removed) by phase

$$\begin{aligned} &\max_{0\leq i<\log(N/L)} \max\left\{\left(i - 1 + \frac{\log\frac{\alpha N}{2^i}}{\log\frac{1}{\rho}} + \frac{\log\frac{N}{L} - i}{C}\right), \log\frac{N}{L} - 1\right\} \\ = &\max_{0\leq i<\log(N/L)} \max\left\{\left(\frac{\log\alpha N}{\log\frac{1}{\rho}} + \frac{\log\frac{N}{L}}{C} + i\left(1 - \frac{1}{\log\frac{1}{\rho}} - \frac{1}{C}\right) - 1\right), \log\frac{N}{L} - 1\right\}, \end{aligned}$$

since a path that is stalled at level i extends to level $i+1$ by phase $i-1+\log(\alpha N/2^i)/\log(1/\rho)$ and its cancellation signals reach level $\log(N/L)$ $(\log(N/L)-i)/C$ phases later. For $C \geq 2$ and $\rho < 1/4$, this expression takes on a maximum value of $\log(N/2L) - 1 + \log(2\alpha L)/\log(1/\rho) + 1/C$. At first, this result seems too good to be true, but stalled real paths catch up to the wavefront very quickly once they get through, and they get through at a very high rate. Hence (for small enough ρ), all real paths get through to the final level along with the wavefront!

Since the number of phases required is basically $\log(N/L)$, the overall time for the algorithm depends mainly on the parameters C and T. By propagating the cancellations at the same time that paths are extended, a single phase can be implemented in $\max(C, 2T+1)$ steps. As long as $\rho < 1/14d$, the algorithm will work for $C \geq 3$. Since $\beta < d-1$, and Lemma 4.2 gives us $\delta = 2\beta/d - 1$, and we need $\rho = (1-\delta)^T < 1/14d$, T must be at least 2. In general, in order to make T small, we need δ to be large.

In order to achieve large δ, we need β to be close to d, which requires α to be small (and consequently L to be large) and d to be large. By using good splitters ($\delta \approx 1$), α small, d large, $C = 5$, and $T = 2$, and replacing each edge with a small constant number of edges, we can obtain a $(5 + \varepsilon) \log N$-step algorithm for routing all the paths. Unfortunately, d and L need to be quite large to achieve this bound. For more reasonable values of d (less than 10) and L (less than 150), we can achieve provable routing times of about $100 \log N$. Fortunately, the algorithms appear to run faster in simulations [19].

It is worth noting that each node needs to keep track of only a few bits of information to make its decisions. This is because only the ith bit of the destination is needed to make a switching decision at level i, and therefore a node at that level looks at this bit, strips it off, and passes the rest of the destination address onward. The path as a whole snakes forward through the network. If it ever gets blocked, the entire snake halts behind it. The implementation details for this scheme are straightforward. Previously, only the AKS sorting circuit was known to achieve this performance for bounded-degree networks, but at a much greater cost in complexity and constant factors. Recently, Leighton and Plaxton have also developed a randomized algorithm for sorting on the butterfly in $O(\log N)$ bit steps [20].

4.4. Routing many paths in a nonblocking fashion on a multi-Beneš network. It is not difficult to implement the circuit-switching algorithm just described on a multi-Beneš network. The main difference between routing through a multi-Beneš network and a multibutterfly network is that in the first half of the multi-Beneš network, a path at a merger input is free to extend to any of the $2d$ neighboring outputs. As the following definition and lemma show, the mergers have an unshared-neighbor property analogous to that of the splitters.

DEFINITION 4.7. *An M-input merger is said to have the (α, δ) unshared-neighbor property if in every subset X of $k \leq \alpha M$ inputs (either up or down or any combination), there are δk nodes in X which have an output neighbor that is not adjacent to any other node in X.*

LEMMA 4.8. *Any merger with the (α, β) expansion property has the (α, δ) unshared-neighbors property, where $\delta = 2\beta/d - 1$, provided that $\beta > d/2$.*

Proof. The proof is essentially the same as that of Lemma 4.2. □

In order to route around existing paths in a multi-Beneš network, we combine the circuit-switching algorithm with the kind of analysis used in §3. To do so, we need to modify the definition of being blocked. A splitter input on level l, $0 \leq l < \log N - \log L$, is blocked if more than $2\beta - d - 1$ of its d up (or down) neighbors on level $l + 1$ are busy or blocked. A merger input on level l, $-\log N + \log L \leq l < 0$, is blocked if more than $4\beta - 2d - 2$ of its $2d$ neighbors on level $l + 1$ are either busy or blocked. Any node that is not blocked is considered to be working.

4.4.1. The subnetwork of working nodes. The following pair of lemmas show that for $\beta > (d + 1)/2$, an unshared-neighbor property is preserved on the working nodes.

LEMMA 4.9. *For $\beta > (d + 1)/2$, the working splitter inputs have an $(\alpha, 1/d)$ unshared-neighbor property.*

Proof. In the proof of Lemma 4.2 we showed that every set X of $k \leq \alpha M$ nodes in an M-input splitter has at least $(2\beta - d)k$ neighbors in the upper and lower outputs with only one neighbor in X. If X is a set of working switches, then at most $(2\beta - d - 1)k$ of these unshared neighbors can be busy or blocked. Thus, at least k of the unshared neighbors must be working. □

LEMMA 4.10. *For $\beta > (d+1)/2$, the working merger inputs have an $(\alpha, 1/d)$ unshared-neighbor property.*

Proof. The proof is similar to that of Lemma 4.9. □

Of course, we must also check that the new blocking definition does not result in any inputs of the multi-Beneš network becoming blocked. This can be done with an argument similar to that in Lemmas 3.1 and 3.2.

LEMMA 4.11. *For $\beta > 2d/3 + 2/3$ and $L > 1/2\alpha(3\beta - 2d - 2)$, less than a 2α fraction of the inputs in any splitter are declared to be blocked. Furthermore, less than an α fraction of the inputs are blocked because of busy and blocked nodes from the upper outputs, and less than an α fraction are blocked because of the lower outputs.*

Proof. The proof is by induction on level number, working backwards from level $\log N - \log L$ to level 0. For the base case, observe that on level $\log N - \log L$ none of the nodes are blocked. Now suppose that αM of the inputs of some M-input splitter are blocked by upper outputs (say), and let $|U|$ be the set of busy or blocked upper outputs. Since the blocked inputs have at least $\alpha\beta M$ neighbors among the upper outputs, and at most $2d - 2\beta + 1$ edges out of each blocked node lead to working nodes, $|U| \geq \alpha M(\beta - (2d - 2\beta + 1)) = \alpha M(3\beta - 2d - 1)$. By induction, however, the number of blocked upper outputs is at most αM and thus $|U| \leq \alpha M + M/2L$. For $L > 1/2\alpha(3\beta - 2d - 2)$, we have a contradiction. □

LEMMA 4.12. *For $\beta > 2d/3 + 2/3$ and $L > 1/2\alpha(3\beta - 2d - 2)$, at most a 2α fraction of the up inputs and at most a 2α fraction of the down inputs in any merger are declared blocked.*

Proof. The proof is similar to that of Lemma 4.11. □

4.4.2. Routing new paths. Once the working nodes have been identified, new paths from the inputs to the outputs of the multi-Beneš network can be established using an algorithm that is essentially the same as the circuit-switching algorithm for multibutterflies described in §4.3. There are two main differences. First, in the multi-Beneš network, only working nodes are used. However, by Lemmas 4.9 and 4.10 the working switches have an $(\alpha, 1/d)$ unshared-neighbors property. Hence, we can run the algorithm of §4.3 with $\delta = 1/d$. Second, routing in the first half of the multi-Beneš network is actually easier than in the second half, which is a multibutterfly, since there is no notion of up or down edges. The goal is simply to get each new path from an input on level $-\log N + \log L$ to any working node on level 0. The algorithm uses placeholders and cancellation signals in the first half in the same way that they are used in the second half.

4.5. Processing incoming calls. Since the working nodes must be identified before new paths can be routed, incoming calls are processed in batches. When a new call originates at an input, it waits until the paths are established for the batch that is currently being processed. When all of the calls in that batch have been established, the working nodes are identified, and then the paths for the new batch are established. Since identifying the working nodes and routing the new paths both take at most $O(\log N)$ bit steps, the time to process each batch is $O(\log N)$ bit steps, and no call waits for more than $O(\log N)$ bit steps before being established, including the time waiting for the previous batch to finish.

5. Extensions.

5.1. Multiparty calls. If all of the parties in a multiparty call are known to a caller at the start of the call, then it is possible to extend the algorithms in §§3 and 4 to route the call from the caller to all of the parties. As a call advances from level 0 to level $\log N$ of the multi-Beneš network, it simply creates branches where necessary to reach the desired output terminals. The bit complexity of the algorithm may increase, however, because more than $O(\log N)$ bits may be needed to specify the set of outputs that the call must reach.

The situation becomes more complicated if parties to a multiparty call are to be added after the call is already underway. One possible solution is to set up paths in the network from the caller to the parties in the call that make multiple passes through the network. To simplify the explanation, let us assume that the input in row i and the output in row i of the multi-Beneš network are actually the same node, for $0 \leq i \leq N-1$. (Thus each input–output can be involved in at most one call.) A multiparty call is established by constructing a binary tree whose root is the caller and whose internal nodes and leaves are the parties in the call. Each node of the binary tree is embedded at an input of the multi-Beneš network, and each edge in the tree from a parent to a child is implemented by routing a path through the multi-Beneš network from the input at which the parent is embedded to the output (which is also an input) at which the child is embedded. To add a new party to the call, we add a new node to the binary tree wherever its depth will be minimum. This ensures that the depth of a tree with l parties will be $O(\log l)$. Since each edge of the binary tree corresponds to a path of length $\log N$ in the network, the path from the root to any other node in the tree has length at most $O(\log^2 N)$ in the network. It's easy to see that a new party can be added in $O(\log^2 N)$ bit steps, but with a little work the time can be brought down to $O(\log N)$ bit steps. One problem with this scheme is that the parties corresponding to internal nodes of the binary tree cannot hang up without also disconnecting all of their descendants. Although this solution is not as elegant as those proposed in [10] for wide-sense generalized nonblocking connectors, no polynomial time-routing algorithms are known for those constructions.

5.2. Multiple calls to the same output. If many parties want to call the same output terminal, then we have two options: merging the callers into a single multiparty call, or giving busy signals to all but one of the callers.

In either case, the first thing to do is to sort the calls according to their destinations. Unfortunately, no deterministic $O(\log N)$-bit-step sorting algorithm is known for the multibutterfly network at present, although $O(\log N)$ word- and bit-step randomized algorithms are known for the butterfly [20, 34]. If a deterministic $O(\log N)$-bit-step algorithm is required, the multibutterfly could be augmented with a sorting circuit such as the AKS sorting circuit [2]. The AKS sorting circuit will provide us with a set of edge-disjoint paths from its inputs to its outputs. If node-disjoint paths are desired, then each 2×2 comparator in the circuit can be replaced by a 2×2 complete bipartite graph. Note that in neither case is the sorting circuit a nonblocking network, since adding new calls at the inputs may alter the sorted order, thus disrupting existing paths. In the remainder of this section, we will use a sorting circuit either in conjunction with a butterfly network to route calls in a rearrangeable fashion, or in conjunction with a multibutterfly to route calls in a nonblocking fashion. In the latter case, the sorting circuit is used only to help compute the routes that the calls take, and not to route the calls themselves.

Once the calls have been sorted, a parallel prefix computation is applied to the sorted list of calls. For each destination, one of the calls is marked as a winner, and the others as losers. For a description of prefix operations, and how they can be implemented in $O(\log N)$ bit steps on a complete binary tree (which is a subgraph of the butterfly), see [16, §1.2].

If it suffices to send a busy signal to all of the callers except one, then these signals can be sent back to the losers along their paths through the sorting circuit, and the winning path can be established (in a nonblocking fashion) in a multibutterfly network.

If the calls are to be merged into a single call, then the next step is to label the winners according to their positions in the sorted order, and to give each loser the label of the winner for its destination. This is also prefix computation.

To route calls in a rearrangeable fashion, we identify the outputs of the sorting circuit with the inputs of a butterfly network. Each call is routed greedily in the butterfly network to the output in the row with the same number as the winner's index. This type of routing problem is called a *packing* problem. Surprisingly, only calls with the same destination will collide during the routing of any packing problem [16, §3.4.3]. After this step, all of the calls to the same destination have been merged into a single call. Since the calls remain sorted by destination, the problem of routing them to their destinations is called a *monotone routing* problem. Any monotone routing problem can be solved with a single pass through two back-to-back butterfly networks without collisions [16, §3.4.3].

To route calls in a nonblocking fashion, we can either assume that all callers are known at the time that a call is established or not. If all of the callers are known, we can route the calls backwards through a multibutterfly from the shared output to each of the inputs of the callers using the first scheme described in §5.1. Otherwise, we can use the second scheme of §5.1 in reverse to route the calls using paths of length $O(\log^2 N)$.

5.3. Removing the distinction between terminals and nonterminals. In this section we generalize the routing algorithm of §4 by removing the distinction between nodes that are terminals and nodes that are not. The algorithm in this section requires $O(\log N)$ word steps, not bit steps. Recall that in the word model, each edge can transmit a word of $O(\log N)$ bits in a single step. The goal of the algorithm is to establish a set of disjoint paths, each of which may start or end at any node in the network. The following similar problem was studied by Peleg and Upfal [27]:

> Given an expander graph, G, K *source* nodes $a_1, \ldots, a_K$ in G, and K *sink* nodes $b_1, \ldots, b_K$ in G, where the sources and sinks are all distinct (i.e., $a_i \neq a_j$ and $b_i \neq b_j$ for $i \neq j$, and $a_i \neq b_j$ for all i and j), construct a path in G from each source a_i to the corresponding sink b_i, so that no two paths share an edge.

Peleg and Upfal presented polylogarithmic-time algorithms for finding K edge-disjoint paths in any n-node expander graph, provided that $K \leq n^\rho$, where ρ is a fixed constant less one. In this section we show that if we are allowed to specify the network (but not the locations of the sources and sinks) then it is possible to construct even more paths. In particular, we describe an n-node bounded-degree network, R, and show how to find K edge-disjoint paths in it in $O(\log n)$ time, provided that $K \leq O(n/\log n)$. Furthermore, we show how to find node-disjoint paths between $\Theta(K)$ of the sources and sinks.

5.3.1. The network. The network R consists of four parts, each of which contains $\log N + 1$ levels of N nodes. Each of the first three parts shares its last level with the first level of the next part, so the total number of levels is $4 \log N + 1$, and the total number of nodes in the network is $n = N(4 \log N + 1)$.

The first part is a set of $\log N + 1$ levels labeled $-2\log N$ through $-\log N$. For $-2\log N \le i < -\log N$, the edges connecting level i to $i+1$ form an N-input merger. Hence, every set of $k \le \alpha N$ nodes on one level has at least $2\beta k$ neighbors on the next level, where α, β, and d are related as in (2.2).

The second part consists of a multibutterfly whose levels are labeled $-\log N$ through 0. The multibutterfly has expansion property (α, β), where α, β, and d are related as in (2.1).

The third and fourth parts are the mirror images of the first and second parts. The levels of these parts are labeled 0 through $2\log N$.

Although any node in R can be chosen to be a source or a sink, it would be more convenient if all of the sources were to reside in the first part, and all the sinks in the fourth. Thus, the node on level $-i$ of the second part, i of the third part, and $2\log N - i$ of the fourth part each have an edge called a *cross* edge to the corresponding node on level $-2\log N + i$ of the first part. Similarly, each node in the fourth part has cross edges to the corresponding nodes in first, second, and third parts. If a node in any part other than the first is chosen to be a source, then its path begins with its cross edge to the first part. If a node in any part other than the fourth is chosen to be a sink, then the path to it ends with a cross edge from the fourth part. At this point, each node in the first part may represent up to four sources, and each node in the fourth part may represent up to four sinks.

5.3.2. Constructing node-disjoint paths. If the paths are to be node-disjoint, then each path must avoid the sources and sinks in the second and third parts as it passes from the first part to the fourth part. To avoid these sources and sinks, we declare them to be *blocked.* We then apply the technique of [17] for tolerating faults in multibutterfly networks to the second and third parts, treating blocked nodes as if they were faulty. The technique of [17] can be summarized as follows. First, any splitter (and all nodes that can be reached from that splitter) that contains more than a $2\alpha(\beta' - 1)$ fraction of blocked inputs is *erased,* meaning that its nodes cannot be used for routing, where $\beta' = \beta - d/4$. Next, working backwards from the outputs to the inputs, a node is declared to be blocked if more than $d/4$ of its up or down neighbors at the next level are blocked (and not erased). (Note that it is not possible for all of a node's up and down neighbors to be erased unless that node is also erased.) Upon reaching the inputs of the network, all the blocked nodes are erased. The switches that are not erased are said to be *working.* The expansion property of the network of working switches is reduced from β to β'.

The following lemmas bound the number of inputs (on levels $-\log N$ and $\log N$) and outputs (on level 0) that are erased in the second and third parts. Note that Lemma 5.1 bounds the number of inputs that are erased, but are not themselves blocked. (All the blocked inputs are erased.) Note also that since the two parts share level 0, the number of erased nodes on that level may be as large as twice the bound given in Lemma 5.2.

LEMMA 5.1. *In addition to the (at most) K blocked inputs, at most $K/(\beta' - 1)$ nonblocked inputs are erased in the second and third parts.*

Proof. This lemma is essentially the same as Lemma 3.3 of [17]. □

LEMMA 5.2. *At most $K/2\alpha(\beta' - 1)$ outputs are erased in each of the second and third parts.*

Proof. This lemma is essentially the same as Lemma 3.1 of [17]. □

In both networks at least $N - O(K)$ of the inputs and $N - O(K)$ of the outputs are left working, where K is the number of sources (and sinks). Suppose that $K \le \gamma N$, where γ is some constant. By choosing γ to be small, we can ensure that at least K of the nodes on level 0 are not erased in either the second or third parts. We call these K nodes the *rendezvous points.* By making β' (and hence d) large, we can also ensure that the number of nodes on levels $-\log N$ and $\log N$ that are erased, but are not themselves sources or sinks, is ϵK, where ϵ can be made to be an arbitrarily small constant.

The reconfiguration technique described in [17] requires off-line computation to count the number of blocked inputs in each splitter. In another paper, Goldberg, Maggs, and Plotkin [12] describe a technique for reconfiguring a multibutterfly on-line in $O(\log N)$ word steps.

The next step is to mark some of the nodes in the first part as blocked. We begin by declaring any node in the first part to be *reserved* if it is a neighbor of a source in the second, third, or fourth part via a cross edge. Now, working backwards from level $-\log N - 1$ to $-2\log N$, a node is declared *blocked* if at least $d/2$ of its $2d$ neighbors at the next level are either sources, sinks, blocked, reserved, or erased. We call a node that is not a source or a sink, and is not reserved, blocked, or erased, a *working* node.

Where did the $d/2$ bound on nonworking neighbors come from? In order to apply the routing algorithm of §4.3, the subnetwork of working nodes must have an (α, δ) unshared-neighbor property. Let β' be the largest value such that the subnetwork of working nodes has an (α, β') expansion property (where (α, β) is the original expansion property of the first part). To show that the subnetwork of working nodes has an (α, δ) unique-neighbors property, we need $\beta' > d/2$. If every working node has at most $d/2$ nonworking neighbors, then the subnetwork of working nodes has expansion property $(\alpha, \beta - d/4)$. (Recall that we multiply the β' parameter by 2 to get the actual expansion in each merger.) Thus $\beta' > \beta - d/4$. If $\beta > 3d/4$, then $\beta' > d/2$. By restricting a working switch to have fewer nonworking neighbors, we could have reduced the required expansion from $3d/4$ down to nearly $d/2$. As the following lemma shows, however, if a working switch can have $d/2$ nonworking neighbors, then we also need $\beta > 3d/4$ in order to ensure that there aren't too many blocked nodes. If we were to allow a working switch to have fewer (or more) than $d/2$ nonworking neighbors, then one of the two "$\beta > 3d/4$" lower bounds would increase, and the network would require more expansion.

LEMMA 5.3. *Let f denote the total number of nodes declared blocked in the first part, let $K \leq \gamma N$ denote the number of sources and sinks, and let ϵK denote the number of nodes on level $-\log N$ that are not sources or sinks, but are erased. Then if $(2+\epsilon)\gamma < (2\beta - 3d/2 - 1)\alpha$, then $f \leq \frac{2+\epsilon}{2\beta - 3d/2 - 1}K$.*

Proof. First, suppose that the total number of blocked nodes in the first part is at most αN. Then the f blocked nodes must have at least $(2\beta - 3d/2)f$ neighbors that are either sources, sinks, blocked, reserved, or erased, since each blocked node has at most $3d/2$ neighbors that are working. Since there are a total of at most K sources and reserved nodes in the first part, at most K sinks, and at most ϵK nodes on level $-\log N$ that are erased, but are not sources or sinks, we have

$$f + 2K + \epsilon K \geq (2\beta - 3d/2)f,$$

which implies that $f \leq \frac{2+\epsilon}{2\beta - 3d/2 - 1}K$.

Otherwise, suppose that there are more than αN blocked nodes in the first part. Let us rank the nodes according to the levels that they appear on (breaking ties within a level arbitrarily), with the nodes on level $\log N - 1$ having highest rank, and those on level $-2\log N$ the lowest. Since the αN blocked nodes with highest rank must have at least $(2\beta - 3d/2)\alpha N$ neighbors that are sources, sinks, blocked, reserved, or erased, we have $\alpha N + 2K + \epsilon K \geq (2\beta - 3d/2)\alpha N$, which is a contradiction for $(2+\epsilon)\gamma < (2\beta - 3d/2 - 1)\alpha$. □

An identical process is applied to the fourth part, with blocked nodes propagating from level $\log N$ to level $2\log N$, and a lemma analogous to Lemma 5.3 can be proven, showing that there are at most $((2+\epsilon)/(2\beta - 3d/2 - 1))K$ blocked nodes in this part.

Because each node in the first part may be reserved by one source in each of the second, third, and fourth parts, it may not be possible for all the sources to establish their paths. If several sources wish to begin their paths at the same node, then one is locally and arbitrarily selected to do so, and the others give up. Since at most four paths start at any node in the first section, at least $K/4$ of the sources are able to begin their paths. Each source then sends a message to the corresponding sink. A message first routes across the row of its source to level $-\log N$ (recall that in every merger there is an edge from each input to the output in the same row), then uses the multibutterfly store-and-forward packet-routing algorithm from [17, 37] to route to the row of its sink on level 0, then routes across that row in the third and fourth parts until it either reaches its sink or reaches the cross edge to its sink. The entire routing can be performed in $O(\log N)$ word steps. Note that we can't use the circuit-switching algorithm of §4.3 here because there may be as many as $\log N$ sinks in a single row. The $K/4$ or more sinks that receive messages then each pick one of these messages (there are at most four), and send an acknowledgment to the source of that message. At least $K/16$ sources receive acknowledgments, and these sources are the ones that will establish paths. A source that doesn't receive an acknowledgment gives up on routing its path.

Some of the nodes at which the remaining sources and sinks wish to begin or end their paths may have been declared blocked. None of these nodes will be used. By making β (and hence d) large, however, the number of blocked nodes in the first and fourth parts, $((2+\epsilon)/(2\beta - 3d/2 - 1))K$, can be made small relative to $K/16$. Thus, we are left with $\Theta(K)$ source-sink pairs.

The paths from the sources and the paths from the sinks are routed independently through the first two and last two parts, respectively. The path from a source a_i then meets the path from the corresponding sink b_i at a rendezvous point r_i on level 0.

The rendezvous points are selected as follows. First, the sources route their paths to distinct nodes on level $-\log N$ in $O(\log N)$ time using the algorithm from §4 on the working switches. Then the sources are numbered according to the order in which they appear on that level using a parallel prefix computation. A parallel prefix computation can be performed in $O(\log N)$ word (or even bit) steps on an N-leaf complete binary tree, and hence also on a butterfly. For a proof, see [16, §1.2]. (Note that although we are treating some of the nodes as if they were faulty, there are no actual faults in the network, so nonworking nodes can assist in performing prefix computations.) The rendezvous points are also numbered according to the order in which they appear on level 0 using another prefix computation.

Next, using a packing operation, a packet representing the ith rendezvous point r_i is routed from r_i to the node in the ith row of level 0. At the same time, a packet representing the ith source a_i is routed from level $-\log N$, where a_i's path has reached so far, to the ith node of level 0. These two routings can be implemented in $O(\log N)$ word (or bit) steps on a butterfly [16, §3.4.3].

Once the packets for a_i and r_i are paired up, a packet is sent back to a_i's node on level $-\log N$ informing it of the position of r_i on level 0. (This is an unpacking operation.) The path for source a_i is then extended from level $-\log N$ to level 0 using the algorithm from §4.3 on the working switches. Then a packet containing the location of r_i is sent from a_i's node on level $-\log N$ to the node on level 0 that is in the same row that b_i lies on in the fourth part. This routing can be performed in $O(\log N)$ word steps using the store-and-forward multibutterfly routing algorithm of [17]. (We can't use the circuit switching algorithm because there may be as many as $\log N$ sinks in the same row.)

In $O(\log N)$ time, the packet works its way across the row from level 0 to b_i, which lies somewhere between levels $\log N$ and $2 \log N$. (Note that although there may be as many as $\log N$ b_i's in the same row, the total time is still at most $O(\log N)$.)

Finally, a path is extended from b_i to any working node on level $\log N$ and from there to r_i using the algorithm of §4.3 on the working switches.

5.3.3. Establishing edge-disjoint paths. It is easier to establish edge-disjoint paths in R than node-disjoint paths. In particular, it is not necessary to apply the technique of [17] for tolerating faults in multibutterflies to the second and third parts of the network as we did in order to establish the node-disjoint paths. The main thing that must be done is to modify the algorithm from §4 for locking down node-disjoint paths in a multibutterfly so that it allows a constant number of edge-disjoint paths to pass through each node. Let r be the maximum number of paths that may pass through a node. In order to replace the unshared-neighbors protocol with one that allows r paths to pass through a node, we define the following r-neighbors property for splitters. Similar definitions hold for mergers, or for pairs of consecutive levels like those in the first and fourth parts of R.

DEFINITION 5.4. *An M-input splitter is said to have an (α, δ) r-neighbors property if in every subset X of $k \leq \alpha M$ inputs, there are subsets X_U and X_D of X such that $X_U \geq \delta k$ and $X_D \geq \delta k$, and every node in X_U (X_D) has at least r up-output (down-output) neighbors, each of which has at most r neighbors in X.*

The following lemma shows that a splitter with a sufficient expansion property also has an r-neighbors property.

LEMMA 5.5. *A splitter with an (α, β) expansion property has an (α, δ) r-neighbors property where*

$$\delta = \frac{\frac{r+1}{r}\beta - \frac{d}{r} - r + 1}{d - r + 1}.$$

Proof. The proof is similar to the proof of Lemma 4.2. Let X be a set of $k \leq \alpha M$ inputs in an M-input splitter, let n_r denote the number of up (down) outputs that have at least one, but at most r, neighbors in X, and let n_+ denote the number of up (down) outputs that have more than r neighbors in X. Then $n_r + n_+ \geq \beta k$, and $n_r + (r+1)n_+ \leq dk$. Solving for n_r yields

$$n_r \geq \left(\frac{r+1}{r}\beta - \frac{d}{r}\right) k.$$

Let δk denote the number of nodes in X with at least r up-output (down-output) neighbors, each of which has at most r neighbors in X. Then $\delta k d + (1-\delta)k(r-1) \geq n_r$, which implies that

$$\delta \geq \frac{\frac{r+1}{r}\beta - \frac{d}{r} - r + 1}{d - r + 1}. \qquad \square$$

The algorithm for routing edge-disjoint paths in a multibutterfly is nearly identical to the algorithm described in §4 for routing node-disjoint paths. First, each node that has at least one path to extend in either the up (or down) direction sends a proposal to each of its output neighbors in the up (down) direction. Then, every output node that receives at most r proposals sends back acceptances to all of those proposals. (Notice that this step limits the number of paths passing through a node to at most r.) Finally, each node that receives enough acceptances to extend *all* of its paths does so. In a network with an (α, δ) r-neighbors property, a constant fraction of the paths on each level are extended at each step. Thus, the time to extend a set of N paths from one level to the next is $O(\log N)$, and the total time to route a set of N paths from the inputs to the outputs is $O(\log^2 N)$. As in §4, this time can be improved to $O(\log N)$ using placeholders and cancellation signals.

Note that for $r \approx \sqrt{d}$, only $\beta \approx \sqrt{d}$ expansion is required in order to have an (α, δ) r-unshared-neighbors property, where $\alpha > 0$ and $\delta > 0$. Since an algorithm for finding edge-disjoint paths can be converted to an algorithm for finding node-disjoint paths by replacing each degree-$2d$ node with a $2d \times 2d$ complete bipartite graph, the algorithm of this section reduces the expansion required for finding either edge- or node-disjoint paths from $\beta > d/2$ to $\beta \approx \sqrt{d}$. The difference is important because explicit constructions of expander graphs are known for $\beta > \sqrt{d}$ [13], but not for $\beta > d/2$. The algorithms for tolerating faults in [17] and the algorithms for routing paths in a nonblocking fashion in this paper still seem to require $\beta > d/2$. Recently, however, Pippenger has shown how to perform all of these tasks using only expansion $\beta > 1$ [31].

In order to use this routing algorithm in network R, we must make one modification. The paths from the sources do not necessarily start on level $-2 \log N$ of the first part. In fact as many as four paths may start at any node in the first part. (Recall that sources in the second, third, and fourth parts start their paths in the first part.) Thus, the routing algorithm must be modified so that a node in the first part sends acceptances to the nodes at the previous level only if it receives at most $r - 4$ proposals. The impact on the performance of the algorithm will be negligible if r is large relative to 4.

Acknowledgments. The authors thank Nick Pippenger for suggesting that a nonblocking network might be constructed by treating busy switches as if they were faulty, and Ron Greenberg and an anonymous referee for suggesting many improvements to the paper.

REFERENCES

[1] W. A. Aiello, F. T. Leighton, B. M. Maggs, and M. Newman, *Fast algorithms for bit-serial routing on a hypercube*, Math. Systems Theory, 24 (1991), pp. 253–271.

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

[3] L. A. Bassalygo and M. S. Pinsker, *Complexity of an optimum nonblocking switching network without reconnections*, Problems Inform. Transmission, 9 (1974), pp. 64–66.

[4] ———, *Asymptotically optimal networks for generalized rearrangeable switching and generalized switching without rearrangement*, Problemy Peredachi Informatsii, 16 (1980), pp. 94–98.

[5] B. Beizer, *The analysis and synthesis of signal switching networks*, in Proc. Symposium on Mathematical Theory of Automata, Brooklyn Polytechnic Institute, Brooklyn, NY, 1962, pp. 563–576.

[6] V. E. Beneš, *Optimal rearrangeable multistage connecting networks*, Bell System Technical J., 43 (1964), pp. 1641–1656.

[7] D. G. Cantor, *On construction of non-blocking switching networks*, in Proc. Symposium on Computer Communication Networks and Teletraffic, Brooklyn Polytechnic Institute, Brooklyn, NY, 1972, pp. 253–255.

[8] D. Dolev, C. Dwork, N. Pippenger, and A. Widgerson, *Superconcentrators, generalizers and generalized connectors with limited depth*, in Proc. 15th Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1983, pp. 42–51.

[9] P. Feldman, J. Friedman, and N. Pippenger, *Non-blocking networks*, in Proc. 18th Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1986, pp. 247–254.

[10] ———, *Wide-sense nonblocking networks*, SIAM J. Discrete Math., 1 (1988), pp. 158–173.

[11] J. Friedman, *A lower bound on strictly non-blocking networks*, Combinatorica, 8 (1988), pp. 185–188.

[12] A. V. Goldberg, B. M. Maggs, and S. A. Plotkin, *A parallel algorithm for reconfiguring a multibutterfly network with faulty switches*, IEEE Trans. Comput., 43 (1994), pp. 321–326.

[13] N. Kahale, *Better expansion for Ramanujan graphs*, in Proceedings of the 32nd Annual Symposium on Foundations of Computer Science, IEEE Computer Society Press, Piscataway, NJ, 1991, pp. 398–404.

[14] C. P. Kruskal and M. Snir, *A unified theory of interconnection network structure*, Theoret. Comput. Sci., 48 (1986), pp. 75–94.

[15] F. T. Leighton, *Parallel computation using meshes of trees*, in 1983 Workshop on Graph-Theoretic Concepts in Computer Science, Trauner Verlag, Linz, 1984, pp. 200–218.

[16] ———, *Introduction to Parallel Algorithms and Architectures: Arrays • Trees • Hypercubes*, Morgan Kaufmann, San Mateo, CA, 1992.

[17] F. T. Leighton and B. M. Maggs, *Fast algorithms for routing around faults in multibutterflies and randomly-wired splitter networks*, IEEE Trans. Comput., 41 (1992), pp. 578–587.

[18] T. Leighton, C. E. Leiserson, and D. Kravets, *Theory of parallel and VLSI computation*, Research Seminar Series Report MIT/LCS/RSS 8, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 1990.

[19] T. Leighton, D. Lisinski, and B. Maggs, *Empirical evaluation of randomly-wired multistage networks*, in Proc. 1990 IEEE International Conference on Computer Design: VLSI in Computers and Processors, IEEE Computer Society Press, Piscataway, NJ, 1990, pp. 380–385.

[20] T. Leighton and G. Plaxton, *A (fairly) simple circuit that (usually) sorts*, in Proc. 31st Annual Symposium on Foundations of Computer Science, IEEE Computer Society Press, Piscataway, NJ, 1990, pp. 264–274.

[21] C. E. Leiserson, *Fat-trees: Universal networks for hardware-efficient supercomputing*, IEEE Trans. Comput., C–34 (1985), pp. 892–901.

[22] G. Lin and N. Pippenger, *Parallel algorithms for routing in non-blocking networks*, in Proc. 3rd Annual ACM Symposium on Parallel Algorithms and Architectures, Association for Computing Machinery, New York, 1991, pp. 272–277.

[23] G. A. Margulis, *Explicit constructions of concentrators*, Problems Inform. Transmission, 9 (1973), pp. 325–332.

[24] G. M. Masson and B. W. Jordan, Jr., *Generalized multi-stage connection networks*, Networks, 2 (1972), pp. 191–209.

[25] D. Nassimi and S. Sahni, *Parallel permutation and sorting algorithms and a new generalized connection network*, J. Assoc. Comput. Mach., 29 (1982), pp. 642–667.

[26] Y. P. Ofman, *A universal automaton*, Trans. Moscow Math. Soc., 14 (1965), pp. 186–199.

[27] D. Peleg and E. Upfal, *Constructing disjoint paths on expander graphs*, in Proc. 19th Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1987, pp. 264–273.

[28] N. Pippenger, *The complexity theory of switching networks*, Ph.D. thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 1973.

[29] ———, *On rearrangeable and nonblocking switching networks*, J. Comput. System Sci., 17 (1978), pp. 145–162.

[30] ———, *Telephone switching networks*, in Proc. Symposia in Applied Mathematics, vol. 26, American Mathematical Society, Providence, RI, 1982, pp. 101–133.

[31] ———, *Self-routing superconcentrators*, in Proc. 25th Annual ACM Symposium on the Theory of Computing, Association for Computing Machinery, New York, 1993, pp. 355–361.

[32] N. Pippenger and L. G. Valiant, *Shifting graphs and their applications*, J. Assoc. Comput. Mach., 23 (1976), pp. 423–432.

[33] N. Pippenger and A. C. Yao, *Rearrangeable networks with limited depth*, SIAM J. Algebraic

Discrete Meth., 3 (1982), pp. 411–417.

[34] J. H. REIF AND L. G. VALIANT, *A logarithmic time sort for linear size networks*, J. Assoc. Comput. Mach., 34 (1987), pp. 60–76.

[35] C. E. SHANNON, *Memory requirements in a telephone exchange*, Bell System Technical J., 29 (1950), pp. 343–349.

[36] J. S. TURNER, *Practical wide-sense nonblocking generalized connectors*, Technical Report WUCS–88–29, Department of Computer Science, Washington University, St. Louis, MO, 1988.

[37] E. UPFAL, *An $O(\log N)$ deterministic packet routing scheme*, in Proc. 21st Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1989, pp. 241–250.

[38] A. WAKSMAN, *A permutation network*, J. Assoc. Comput. Mach., 15 (1968), pp. 159–163.

SIAM J. COMPUT.
Vol. 25, No. 3, pp. 626–647, June 1996

THE LINKAGE OF A GRAPH*

LEFTERIS M. KIROUSIS† AND DIMITRIS M. THILIKOS‡

Abstract. The linkage of a graph is defined to be the maximum min-degree of any of its subgraphs. It is known that the linkage of a graph is equal to its width: for an arbitrary linear ordering of the vertices of the graph, consider the maximum, with respect to any vertex v, of the number of vertices connected with v and preceding it in the ordering; the width of the graph is the minimum of these maxima over all possible linear orderings. Width has been used in artificial intelligence in the context of constraint satisfaction problems (CSPs). A more general notion is defined by considering not the number of vertices preceding and connected with v but rather the least number of vertices preceding and connected with any cluster of at most j consecutive vertices extending to the right up to v (j is a given integer). The graph parameter thus defined is called j-width. No efficient algorithm was known for computing the j-width. In this paper, we introduce a graph parameter depending on j that refers to the subgraphs of the graph and generalizes the notion of linkage. We prove the min–max theorem that this graph parameter, which we call j-linkage, is equal to j-width, and we then give a polynomial-time algorithm for computing it (for constant j). We also find *tight* lower and upper bounds for the j-linkage (equivalently, the j-width) of graphs with given numbers of vertices and edges. It is interesting to note that a lower bound for the width of a graph had been found by Erdös; as we show, however, that bound is not tight. Moreover, we prove that our lower bound for width is also a *tight* lower bound for treewidth, pathwidth, and bandwidth, graph parameters that may be arbitrarily larger than width. Finally, we show that computing the j-linkage is a P-complete problem, whereas we prove that approximating it is a threshold problem: it is in NC for approximation factors $< 1/(2j)$, and it is P-complete for approximation factors $> 1/2$.

Key words. width parameters of a graph, linkage of a graph, backtrack-free search, extremal graph properties, algorithms in NC, P-complete problems

AMS subject classifications. 68Q15, 68Q20, 68Q22, 68Q25, 68R10, 05C35, 05C85

1. Introduction.

Let $G = (V, E)$ be an undirected graph without multiple edges or loops. Let $n = |V|$ and $e = |E|$.

The linkage of G is defined to be the maximum min-degree of any of the subgraphs of G (the min-degree of a subgraph is the least degree of any of its vertices; the degree of a vertex is taken relative to the subgraph).

The width of G is defined to be the the minimum, over all linear orderings of the vertices of G, of the maximum, with respect to any vertex v, of the number of vertices connected with v and preceding it in the linear ordering. In [14], it was proved that the width of a graph is equal to its linkage (see also [7]). The term linkage was introduced by Freuder in [7]; however, the corresponding notion dates back to Szekeres and Wilf [19] and Matula [12]. Using the aforementioned equality of the two parameters, it can be proved that they can be computed in polynomial time.

Erdös [6] proved that every graph G has a subgraph with min-degree at least equal to the density $\lceil e/n \rceil$ of G. Therefore, the density is a lower bound for the linkage (equivalently, width) of graphs with given numbers of edges and vertices.

Given a positive integer j, if in the definition of width we consider not the number of vertices preceding and connected with v but rather the least number of vertices preceding and connected with any cluster of at most j consecutive vertices extending

* Received by the editors September 1, 1993; accepted for publication (in revised form) October 26, 1994. This research was partially supported by the European Union's ESPRIT Basic Research project ALCOM II (contract 7141).

† Department of Computer Engineering and Informatics, Patras University, Rion, 265 00 Patras, Greece; Computer Technology Institute, P.O. Box 1122, 261 10 Patras, Greece (kirousis@cti.gr).

‡ Department of Computer Engineering and Informatics, Patras University, Rion, 265 00 Patras, Greece; Computer Technology Institute, P.O. Box 1122, 261 10 Patras, Greece (sedthilk@cti.gr).

to the right up to v, we get a graph parameter known as j-width (see next section for formal definitions). This notion is due to Freuder [8].

Freuder [7] related width to backtrack-free search for a solution of a constraint satisfaction problem (CSP). Roughly, by Freuder's result, when the level of local consistency (strong consistency) of the constraint graph of a CSP exceeds its width, there is a backtrack-free search for a globally consistent solution. Corresponding results that concern backtrack-bounded search hold in relation to j-width [8]. However, the algorithm given by Freuder for computing the j-width has worst-case complexity of the order of $n!$ (for constant j). To our knowledge, no polynomial-time algorithm for computing the j-width was known.

In this paper, we introduce a graph parameter that we call j-linkage. Given a subgraph $H = (V_H, E_H)$ of G, consider all nonempty sets $S \subseteq V_H$ with cardinality at most j and find the minimum, over all such subsets S, of the number of vertices in $V_H - S$ that are adjacent to vertices in S; we then find the maximum, over all subgraphs H of G, of the corresponding minima. This maximum is by definition the j-linkage of the graph (see next section for formal definitions). Notice that the linkage of G is equal to its 1-linkage. We prove the min–max result that the j-width of a graph is equal to its j-linkage. We also give a polynomial-time sequential algorithm for computing the j-linkage when j is a fixed constant. We thus provide an efficient way for computing the j-width.

Furthermore, we find a *tight* lower bound for the j-linkage of graphs with given numbers of vertices and edges (for any $j \geq 1$). In contrast, we show that the aforementioned lower bound by Erdös (for $j = 1$) is not tight. Moreover, we prove that our lower bound for width is also a *tight* lower bound for treewidth, pathwidth and bandwidth, graph parameters that may be arbitrarily larger than width. There is extensive literature on lower bounds for these latter parameters [15–17, 20], but no result referring to tightness was known. To obtain the lower bound for j-linkage, we consider graphs that have the extremal property that their j-width increases by the addition of even one arbitrary edge. We then find the maximum number of edges that such an extremal graph may have, when its j-linkage and its number of vertices is k and n, respectively. Call this maximum max-$E(n, j, k)$. We prove that for an arbitrary graph with e edges and n vertices, respectively, the least k for which max-$E(n, j, k) \geq e$ is the required lower bound for the j-linkage. We show that this bound is tight by constructing the corresponding extremal graphs. We believe that the method of using extremal graph properties to compute tight bounds is interesting in itself.

Using a method similar to the one described above, we obtain a *tight upper* bound for the j-linkage. It should be mentioned that one can find a nontight upper bound with simpler computations. However, for certain values of j, the difference of this upper bound from the tight one that we provide is $\Theta(e^{1/2})$.

Finally, we study the parallel complexity of computing the j-linkage. For constant j, we show that the problem of determining whether j-linkage is no more than k is P-complete for a any fixed $k \geq 2$, whereas it is in NC for $k \leq 1$. We also consider the parallel complexity of approximating the j-linkage. We show that this problem is of the threshold type: it is in NC for approximation factors $< 1/(2j)$, and it is P-complete for approximation factors $> 1/2$.

In summary: (1) we study the sequential and parallel complexity of computing the j-width (alternatively, j-linkage), we resolve the question of efficiency in the sequential case, and we prove positive and negative results for the parallel case; (2) we provide optimal upper and lower estimates for the j-width that, when $j = 1$, improve previous results, and we give connections with other graph parameters.

2. Graph-theoretic properties and the elimination algorithm.

In this section, we first give formal definitions for the concepts we use, and we then prove that the j-width of a graph is equal to its j-linkage. Finally, we give a polynomial-time sequential algorithm for computing the j-linkage.

2.1. Definitions.

Let l be a *layout* of a graph $G = (V, E)$, i.e., a linear ordering $v_1, \ldots, v_n$ of its vertices. The *width* with respect to l of a set $S = \{v_{i-k+1}, \ldots, v_i\}$ of k consecutive vertices (notationally $\text{width}_l(S)$) is defined to be the number of vertices in the set $\{v_1, \ldots, v_{i-k}\}$ adjacent to vertices in S $(1 \leq k \leq i \leq n)$.

Informally, the width of S with respect to l is the number of vertices preceding S and adjacent to elements of S.

Now, let j be an integer such that $1 \leq j \leq n$.

The *j-width* of a vertex v_i with respect to l is the least width of any set of at most $\min(i, j)$ consecutive vertices extending to the right up to v_i, i.e.,

$$j\text{-width}_l(v_i) = \min\{\text{width}_l(v_{i-k+1}, \ldots, v_i) \colon k = 1, \ldots, \min(i, j)\}.$$

The *j-width* of G with respect to l is defined to be the maximum of $j\text{-width}_l(v_i)$ over all vertices v_i of G.

Finally, the j-width of G (not dependent on a specific layout) is defined to be the minimum of the j-widths of G with respect to any of the $n!$ layouts of G.

The 1-width of G is simply called the width of G.

We now introduce the notion of the j-linkage of a graph. The related definitions follow.

External degree of a nonempty set S of vertices of a subgraph H of G (notationally $\text{ext-degree}_H(S)$) is the number of vertices of H that do not belong to S and are adjacent to an element of S.

The *j-min-degree* of a subgraph H of G is the minimum $\text{ext-degree}_H(S)$ over all sets S of vertices of H with $1 \leq |S| \leq j$. Obviously, the 1-min-degree of a subgraph is the least degree of its vertices.

The *j-linkage* of G is the maximum j-min-degree of any subgraph of G.

Obviously, the 1-linkage of G is the maximum min-degree of any of its subgraphs. The 1-linkage of G is simply called the linkage of G.

2.2. j-width equals j-linkage.

THEOREM 2.1. *For any graph $G = (V, E)$, the j-width of G is equal to its j-linkage.*

Proof. Let $j\text{-linkage}(G) = \lambda$. We first show that $j\text{-width}(G) \geq \lambda$. From the definition of linkage, it follows that there is an induced subgraph $H = (V_H, E_H)$ of G such that for any set S of vertices of V_H with $1 \leq |S| \leq j$, $\text{ext-degree}_H(S) \geq \lambda$. Now, consider an arbitrary layout l of G. Let v_i be the last vertex in this layout that belongs to V_H. We claim that the j-width of v_i with respect to l is at least λ. Indeed, let S_0 be the intersection V_H with an arbitrary set $\{v_{i-k+1}, \ldots, v_i\}$ of k consecutive vertices extending to the right up to v_i, where k is an arbitrary integer less than or equal to $\min(i, j)$. Obviously, S_0 is a nonempty subset of V_H of cardinality at most j. Moreover, all vertices of $V_H - S_0$ that are adjacent to vertices in S_0 must belong to $\{v_1, \ldots, v_{i-k}\}$ because v_i was chosen to be the last with respect to l vertex of V_H. Therefore, because $\text{ext-degree}_H(S_0) \geq \lambda$, we get, as we claimed, that the j-width of v_i with respect to l is at least λ. Because l was arbitrary, it follows that the j-width$(G) \geq \lambda = j$-linkage(G).

We will now show that j-width$(G) \leq \lambda$. From the definition of linkage, it also follows that for any induced subgraph $H = (V_H, E_H)$ of G, there is a set S of vertices of V_H with $1 \leq |S| \leq j$ and ext-degree$_H(S) \leq \lambda$. In the next paragraph, we give a construction of a layout l of G such that for any vertex v of G, j-width$_l(v) \leq \lambda$. Once such an l is constructed, the required inequality easily follows.

The construction of l will be given in a last-first fashion, i.e., we proceed from defining the last vertices in l towards defining the first elements in it. Because j-linkage$(G) = \lambda$ and since G is a subgraph of itself, we get that j-min-degree$(G) \leq \lambda$. Therefore, there exists a set S_0 of vertices of G such that $1 \leq |S_0| \leq j$ and ext-degree$_G(S_0) \leq \lambda$. We place the vertices of S_0 (in any arbitrary order) as the last vertices in the layout under construction. Notice that every subset of S_0 is adjacent with at most λ vertices among the ones that will be later placed in the layout. Therefore, any vertex in S_0 has j-width with respect to the layout l at most λ. We now consider the subgraph H induced by the set of vertices $V - S_0$. Again because j-min-degree$(H) \leq \lambda$, there must exist a subset S_1 of vertices of H such that $1 \leq |S_1| \leq j$ and ext-degree$_H(S_1) \leq \lambda$. As a next step towards defining l, we place the elements of S_1 (in any arbitrary order) to the left of the vertices of S_0. By the same argument as above, any vertex in S_1 must have j-width (with respect to the layout under construction) at most λ. Continuing recursively in the same way until the vertices of G are exhausted, we obtain the required layout of G. □

In the rest of the paper, we use the terms j-linkage and j-width interchangeably.

2.3. The elimination algorithm. In this subsection, we describe an algorithm that given a k $(0 \leq k \leq n-j-1)$ finds the unique maximal vertex-induced subgraph of $G = (V, E)$ with j-min-degree at least $k+1$, in the case that there is such a subgraph; otherwise it returns the empty set. The complexity of the algorithm, for fixed j, is polynomial in n. This algorithm can be used to compute the j-linkage of a graph in time polynomial in n. We call this algorithm the (k, j)-Elimination Algorithm or just k-elimination when no confusion may arise. The graph that the algorithm returns on input G will be denoted by $G^{(k,j)}$.

```
(k, j)-ELIMINATION ALGORITHM
do while $\exists S \subseteq V$ such that $1 \leq |S| \leq j$ and ext-degree$_G(S) \leq k$
    begin
    Let $S$ be a set of vertices with $1 \leq |S| \leq j$ and ext-degree$_G(S) \leq k$
    $V := V - S$
    if $V \neq \emptyset$ then $G :=$ the graph induced by $V$
                else $G := \emptyset$
    end
return $G$
```

THEOREM 2.2. *The (k, j)-Elimination Algorithm finds a maximal vertex-induced subgraph of G that has j-min-degree at least $k+1$ in the case that there is such a subgraph; otherwise, it returns the empty set. Moreover, such a maximal subgraph is unique.*

Proof. Suppose that the algorithm outputs a nonempty subgraph $H = (V_H, E_H)$. Because V_H does not contain any nonempty set S with ext-degree$_H(S) \leq k$, we have that j-min-degree$(H) > k$. Now suppose, towards a contradiction, that there is a proper supergraph $R = (V_R, E_R)$ of H, with j-min-degree$(R) > k$.

Let S be the first set of vertices containing vertices of R that is removed, according to the algorithm, from the set of vertices of G (such a set exists because R is a proper supergraph of H). Notice that by the definition of S, no vertex of R is removed before the set S is discarded.

Notice now that because j-min-degree$(R) > k$, ext-degree$_R(S \cap V_R) > k$. But then it easily follows that the external degree of S with respect to any supergraph of R whose vertices contain S is strictly greater than k. This contradicts the fact that S is discarded by the algorithm at a stage when all the vertices of R are still present.

To show the uniqueness of such a maximal subgraph, repeat the above argument taking R not to be a proper supergraph of H but rather a graph with $V_R - V_H \neq \emptyset$. □

It is easy to see that in order to compute the j-linkage of G, we have simply to find the smallest k for which the k-elimination returns the empty graph. The complexity of this algorithm is $O(n^{j+2})$.

3. Extremal graph properties and applications in computing tight bounds.

In this section, we find tight lower and upper bounds for the j-linkage of graphs with given numbers of vertices and edges.

3.1. Lower bound.

To compute a lower bound for j-linkage, we consider the graphs whose j-linkage increases by the addition of even one arbitrary edge. We then find the maximum number of edges that such an extremal graph may have when its j-linkage and its number of vertices are k and n, respectively (see also [11] for a similar result for the case $j = 1$). Call this maximum max-$E(n, j, k)$. Next, we prove that for an arbitrary graph with e edges and n vertices, respectively, the unique k for which max-$E(n, j, k-1) < e \leq$ max-$E(n, j, k)$ is the required lower bound for j-linkage.

LEMMA 3.1. *Let G be a graph with j-linkage at most k. Suppose that G has the property that any graph obtained from G by adding one more edge (on the same set of vertices) has j-linkage equal to $k + 1$. Then the number of edges of G is at most*

$$\text{max-}E(n,j,k) = \binom{k}{2} + (n-k)k - \frac{1}{2}(n-k) + \frac{1}{2}(\lfloor (n-k)/j \rfloor j^2 + ((n-k) \bmod j)^2). \tag{1}$$

Moreover, for any $n \geq 1$, for any $j = 1, \ldots, n$ and any k such that $0 \leq k \leq n - j$, there exists at least one graph with n vertices, max-$E(n, j, k)$ *edges, and j-linkage equal to k and such that if one more edge is added to it (on the same set of vertices), its j-linkage increases to $k + 1$.*

Proof. To prove the above lemma, we use a technical result about extreme points.

TECHNICAL LEMMA. *Given real numbers Z and D, among all sequences $A_1, A_2, \ldots, A_r$ of real numbers such that*

- *r is an arbitrary integer ≥ 1,*
- *$A_1 + \cdots + A_r = Z$, and*
- *$A_i \leq D$, for all $i = 1, \ldots, r$,*

the maximum value that the quantity $\sum_{i=1}^{r} A_i^2$ attains is $\lfloor Z/D \rfloor D^2 + (Z - \lfloor Z/D \rfloor D)^2$. This value is obtained for $r = \lceil Z/D \rceil$, $A_i = D$ (for all $i = 1, \ldots, \lfloor Z/D \rfloor$), and (only in case Z/D is not an integer) $A_{\lceil Z/D \rceil} = Z - \lfloor Z/D \rfloor D$.

The Technical Lemma follows from the following claim applied for $W = Z/D$ and $y_i = A_i/D$ (for notational convenience, we set $\{W\} = W - \lfloor W \rfloor$).

CLAIM. *For any integer $n > 0$, real number $W \geq 0$ and sequence of reals y_i, $i = 1, \ldots, n$, such that $0 \leq y_i \leq 1$, if $\sum_{i=1}^{n} y_i = W$, then $\sum_{i=1}^{n} y_i^2 \leq \lfloor W \rfloor + \{W\}^2$.*

Proof. We use induction on n. For $n = 1$, we can see that

- if $y_1 = 1$, the claim holds because $1^2 = \lfloor 1 \rfloor + \{1\}^2$;
- if $0 \leq y_1 < 1$, the claim holds because $y_1^2 = \lfloor y_1 \rfloor + \{y_1\}^2$.

Suppose that the proposition holds for $n = k$. We are going to prove that it also holds for $n = k + 1$. So we have to prove that for $n = k + 1$, any real number $W > 0$, and any sequence y_i, $i = 1, \ldots, k + 1$, such that $0 \leq y_i \leq 1$, if $\sum_{i=1}^{k+1} y_i = W$, then $\sum_{i=1}^{k+1} y_i^2 \leq \lfloor W \rfloor + \{W\}^2$.

Indeed, because $\sum_{i=1}^{k+1} y_i = W$, we have that $\sum_{i=1}^{k} y_i = W - y_{k+1}$. By applying the induction hypothesis, we have that

$$\sum_{i=1}^{k} y_i^2 \leq \lfloor W - y_{k+1} \rfloor + \{W - y_{k+1}\}^2.$$

Therefore,

$$\sum_{i=1}^{k} y_i^2 + y_{k+1}^2 \leq \lfloor W - y_{k+1} \rfloor + \{W - y_{k+1}\}^2 + y_{k+1}^2. \tag{2}$$

We distinguish two cases:

1. $y_{k+1} \leq \{W\}$ and
2. $y_{k+1} > \{W\}$.

In the first case, we have that

$$\lfloor W - y_{k+1} \rfloor = \lfloor W \rfloor, \text{ and therefore} \tag{3}$$

$$\{W - y_{k+1}\} = \{W\} - y_{k+1}. \tag{4}$$

From relations (2)–(4), we have that

$$\begin{aligned}\sum_{i=1}^{k+1} y_i^2 &\leq \lfloor W \rfloor + \{W\}^2 + y_{k+1}^2 - 2\{W\}^2 y_{k+1} + y_{k+1}^2 \\ &= \lfloor W \rfloor + \{W\}^2 + 2y_{k+1}(y_{k+1} - \{W\}) \\ &\leq \lfloor W \rfloor + \{W\}^2.\end{aligned}$$

In the second case, we have that

$$\lfloor W - y_{k+1} \rfloor = \lfloor W \rfloor - 1, \text{ and therefore} \tag{5}$$

$$\{W - y_{k+1}\} = 1 + \{W\} - y_{k+1}. \tag{6}$$

From relations (2), (5), and (6), we have that

$$\begin{aligned}\sum_{i=1}^{k+1} y_i^2 &\leq \lfloor W \rfloor - 1 + 1 + \{W\}^2 + y_{k+1}^2 + 2\{W\} - 2y_{k+1} - 2\{W\}y_{k+1} + y_{k+1}^2 \\ &= \lfloor W \rfloor + \{W\}^2 + 2y_{k+1}(y_{k+1} - 1) - 2\{W\}(y_{k+1} - 1) \\ &= \lfloor W \rfloor + \{W\}^2 + 2(y_{k+1} - \{W\})(y_{k+1} - 1) \\ &\leq \lfloor W \rfloor + \{W\}^2.\end{aligned}$$

Therefore, the Claim holds for $n = k + 1$. That proves the Claim and concludes the proof of the Technical Lemma. □

We now proceed with the proof of Lemma 3.1.

Proof of Lemma 3.1. By hypothesis, we have that G satisfies the following two properties:

1. j-linkage$(G) = k$ and
2. j-linkage$(G') = k+1$ where G' is any graph obtained from G by adding one more edge (on the same set of vertices).

Property 1 above implies that if we apply the k-elimination to G, all vertices of G will be removed, whereas by property 2, the k-elimination does not remove all vertices of a graph G' obtained by adding to G at least one more edge (on the same set of vertices).

Now suppose that the k-elimination is applied to G and let $S_0, \dots, S_l$ be the sets of vertices successively removed from V by the algorithm. Let V_i $(i = 1, \dots, l)$ be the set of vertices of G remaining after the removal of S_{i-1}. For notational convenience, let V_0 be the initial set of vertices of G. Let also G_i be the graph induced by V_i. Let q be the first step such that $|V_{q+1}| \leq k+j$ (notice that then $|V_{q+1}| > k$).

Observe first that for all $i = 0, \dots, q$, ext-degree$_{G_i}(S_i) = k$. Indeed, otherwise, we could add one more edge to the graph G without increasing its j-linkage. What's more, observe that for the same reason *all* vertices of each S_i are connected with the same set of exactly k vertices in V_{i+1}. Moreover, again for the same reason, the graphs induced by S_i $(i = 1, \dots, q)$ and V_{q+1} are cliques. Now, for notational convenience, let $|V_{q+1}| = k + \rho$, where $1 \leq \rho \leq j$. From the above observations, it follows that the number of edges of G is

$$\sum_{i=0}^{q}\left(|S_i|k + \binom{|S_i|}{2}\right) + \binom{k+\rho}{2}.$$

Now observe that $\binom{k+\rho}{2} = \binom{k}{2} + \binom{\rho}{2} + \rho k$. Again, for notational convenience, let $A_i = |S_i|$ (for $i = 0, \dots, q$) and $A_{q+1} = \rho$. We can easily see that $A_i \leq j$ (for $1 \leq i \leq q+1$) and that $\sum_{i=0}^{q+1} A_i = n - k$.

So, finally,

$$\begin{aligned}
&\sum_{i=0}^{q}\left(|S_i|k + \binom{|S_i|}{2}\right) + \binom{k+\rho}{2}\\
&= \sum_{i=0}^{q}\left(A_ik + \binom{A_i}{2}\right) + A_{q+1}k + \binom{k}{2} + \binom{A_{q+1}}{2}\\
&= \sum_{i=0}^{q+1} A_ik + \frac{1}{2}\sum_{i=0}^{q+1}(A_i^2 - A_i) + \binom{k}{2}\\
&= (n-k)k - \frac{1}{2}(n-k) + \frac{1}{2}\sum_{i=0}^{q+1} A_i^2 + \binom{k}{2}.
\end{aligned}$$

By applying the Technical Lemma for $Z = n - k$ and $D = j$, we obtain that the quantity $\sum_{i=1}^{q+1} A_i^2$ has maximum value $\lfloor (n-k)/j \rfloor j^2 + ((n-k) \bmod j)^2$. Therefore, we conclude that a graph with properties 1 and 2 has at most

$$\binom{k}{2} + (n-k)k - \frac{1}{2}(n-k) + \frac{1}{2}((\lfloor (n-k)/j \rfloor)j^2 + ((n-k) \bmod j)^2)$$

edges. This completes the proof of the first statement of the lemma.

To prove the second statement of the lemma, given n, j, and k $(n \geq 1, j = 1, \dots, n,$ $0 \leq k \leq n-j)$, we construct a graph G with max-$E(n,j,k)$ edges, n vertices, j-linkage

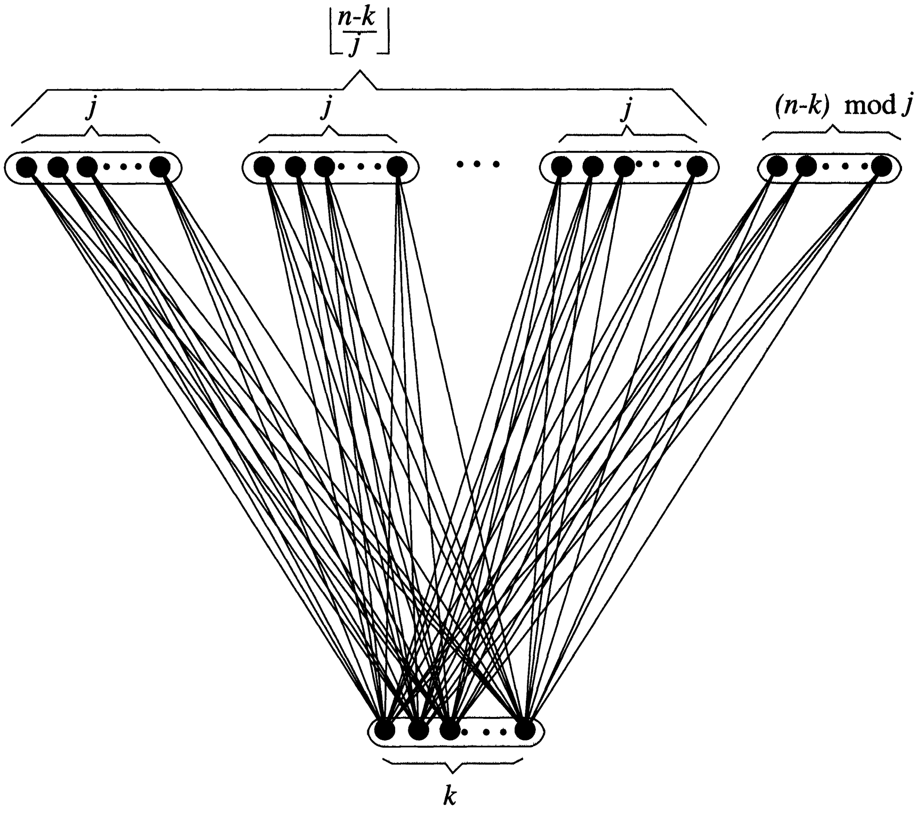


FIG. 1. *The construction of the second part of Lemma* 3.1.

equal to k, and such that with one more edge the j-linkage of G becomes $k+1$. Towards this, consider one clique with k vertices, $\lfloor (n-k)/j \rfloor$ cliques with j vertices each, and one clique with $(n-k) \bmod j$ vertices. Connect all the vertices of the k-clique with all the vertices of all other cliques. Let G be the graph thus constructed (see Fig. 1). Observe that G has $k+\lfloor (n-j)/j \rfloor j+((n-k) \bmod j) = n$ vertices and max-$E(n,j,k)$ edges. Also, observe that there is no nonempty set S with at most j vertices and with ext-degree$_G(S) < k$. Moreover, a k-elimination discards all vertices of G. Therefore, j-linkage$(G) = k$. Also, it easy to see that if we add one more edge to G, then a k-elimination returns a nonempty set. Therefore, we also proved the second statement of the lemma. □

We now state the following two easy lemmas.

LEMMA 3.2. *A graph G with j-linkage$(G) \leq k$ has at most* max-$E(n,j,k)$ *edges.*

LEMMA 3.3. *The function* max-$E(n,j,k)$ *is strictly increasing with respect to k.*

Proofs. The proofs of both the above lemmas are straightforward. Indeed, to prove the first, proceed recursively as follows: as long as a new edge can be added to G without increasing its j-linkage strictly above k, add such an edge. The graph thus obtained has at most max-$E(n,j,k)$ edges. For the second lemma, consider a graph with max-$E(n,j,k)$ edges. As long as a new edge can be added to this graph without increasing its j-linkage strictly above $k+1$, add such an edge; notice that the resulting graph has strictly more edges than the original. □

Now, given natural numbers $n \geq 1$, $j = 1, \ldots, n$, and e such that max-$E(n,j,0) <$

$e \leq \binom{n}{2}$ let $L(e,n,j)$ (or just L when no confusion may arise) be the unique integer that satisfies the following relation:

$$\text{max-}E(n,j,L(e,n,j)-1) < e \leq \text{max-}E(n,j,L(e,n,j)). \tag{7}$$

The uniqueness of the integer $L(e,n,j)$ follows from Lemma 3.3. Moreover, for completeness, we set $L(e,n,j) = 0$, for any $n \geq 1, j = 1,\ldots,n$, and e such that $0 \leq e \leq \text{max-}E(n,j,0)$.

THEOREM 3.4. *$L(e,n,j)$ is a lower bound for the j-linkage of graphs with e edges and n vertices. Moreover, for arbitrary n and e such that $n \geq 1$ and $0 \leq e \leq \binom{n}{2}$, there is a graph G with e edges and n vertices such that j-linkage$(G) = L(e,n,j)$ (i.e., $L(e,n,j)$ is a* tight *lower bound for the j-linkage).*

Proof. We give the proof only if $\text{max-}E(n,j,0) < e \leq \binom{n}{2}$. The case where $e \leq \text{max-}E(n,j,0)$ is similar and easier. The fact that $L(e,n,j)$ is a lower bound follows easily from Lemma 3.2 and relation (7). In order to prove that $L(e,n,j)$ is also a tight bound, we construct, for given e, n, and j, a graph G with e edges, n vertices, and j-linkage equal to $L(e,n,j)$.

First, by applying the construction in the proof of the second statement of Lemma 3.1 for $k = L-1$, we obtain a graph—call it H—such that j-linkage$(H) = L-1$ and such that any graph obtained from H by adding one more edge (on the same set of vertices) has j-linkage equal to L. By construction, the graph H has $\text{max-}E(n,j,L-1)$ edges. Let $d = e - \text{max-}E(n,j,L-1)$. From relations (1) and (7), we have that $1 \leq d \leq n - L - ((n-L) \bmod j)$. We will construct G by adding to H d new edges. We distinguish the following two cases:

- *Case* 1. $(n-(L-1)) \bmod j = 0$.
- *Case* 2. $(n-(L-1)) \bmod j \neq 0$.

In the first case, among the j-cliques that were used to construct H, we take an arbitrary one and connect one arbitrary vertex of it with d vertices of the remaining j-cliques (these d vertices are again arbitrarily chosen). Such a selection of d vertices is possible because (i) $d \leq n - L - ((n-L) \bmod j)$, (ii) $(n-L) \bmod j = j-1$ (because $(n-(L-1)) \bmod j = 0$), and (iii) the remaining j-cliques have exactly $n-(L-1)-j = n-L-(j-1)$ vertices. It is now easy to see that the graph G has j-linkage equal to L.

In the second case (see Fig. 2), we construct G by connecting an arbitrary vertex of the $((n-(L-1)) \bmod j)$-clique of H with d vertices chosen from the remaining j-cliques of H. The number of vertices of the j-cliques of H are $n-(L-1)-((n-(L-1)) \bmod j)$. The selection of d vertices is possible because (i) as we noticed before, $d \leq n-L-((n-L) \bmod j)$ and (ii) obviously, $n-L-((n-L) \bmod j) \leq n-(L-1)-((n-(L-1)) \bmod j)$. Finally, it is again easy to see that the graph G has j-linkage equal to L. □

Since $\text{max-}E(n,j,L)$ is given by a closed formula, the number $L(e,n,j)$ can be efficiently computed. Moreover, as the following theorem shows, we can find an almost closed formula for computing $L(e,n,j)$. Indeed for $j=1$, the two bounds provided by the theorem below lead to an exact value for L, whereas, if j is arbitrary, the difference between the two bounds is $O(j)$.

THEOREM 3.5. *For any e, n, and j such that $n \geq 1$, $1 \leq j \leq n$, and $0 \leq e \leq \binom{n}{2}$, we have that*

$$L(e,n,j) \geq \frac{1}{2}\left(2n - j - \sqrt{(2n-j)^2 + 4n(j-1) - 8e}\right) \quad \textit{and}$$

$$L(e,n,j) < \frac{1}{2}\left(2n - j + 2 - \sqrt{(2n-j)^2 + 4n(j-1) - 8e + j^2 - 1}\right).$$

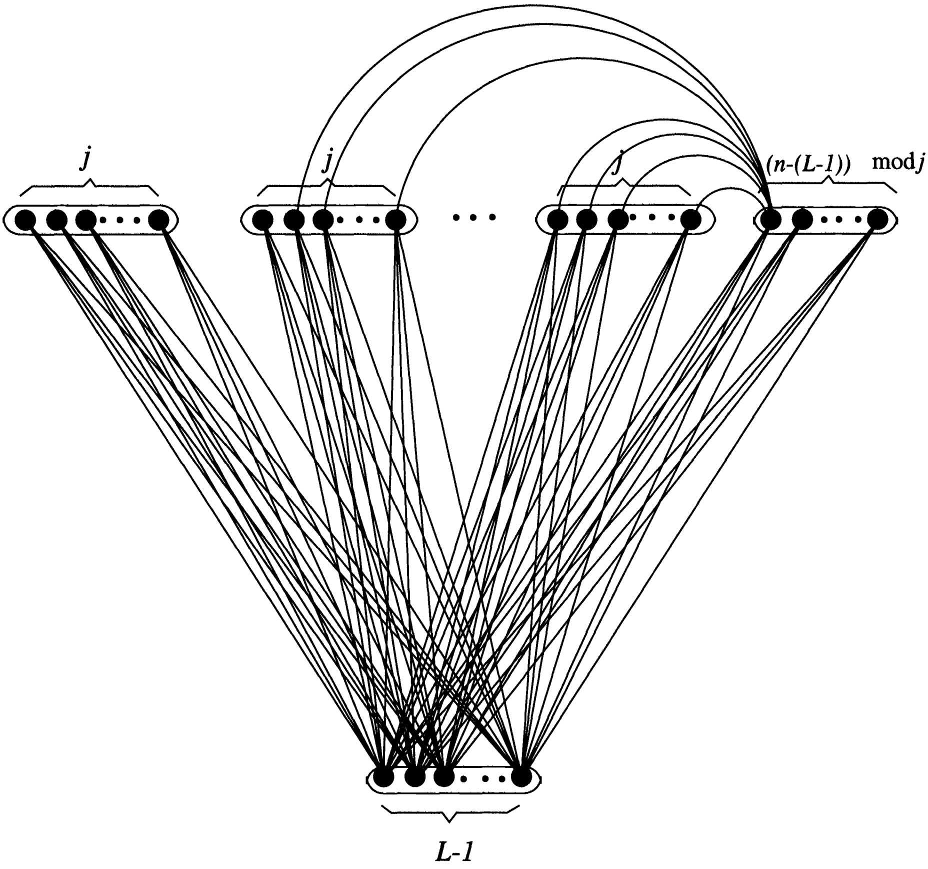


FIG. 2. *The construction of the second part of Theorem* 3.4.

Proof. For notational convenience, define

$$\sigma_L = (n-L) \bmod j \text{ and}$$
$$Q_L = \lfloor (n-L)/j \rfloor j^2 + ((n-L) \bmod j)^2.$$

Observe that

$$\text{max-}E(n,j,L) = \binom{L}{2} + (n-L)L - \frac{1}{2}(n-L) + \frac{1}{2}Q_L. \tag{8}$$

Also observe that $0 \leq \sigma_L \leq j-1$. We now obtain

$$\begin{aligned} Q_L &= \lfloor (n-L)/j \rfloor j^2 + \sigma_L^2 \\ &\leq \lfloor (n-L)/j \rfloor j^2 + \sigma_L j \\ &= (\lfloor (n-L)/j \rfloor j + \sigma_L) j \\ &= (n-L)j. \qquad (9) \end{aligned}$$

Moreover,

$$Q_L = \lfloor (n-L)/j \rfloor j^2 + \sigma_L^2$$

$$
\begin{aligned}
&= (n-L)j - \sigma_L j + \sigma_L^2 \\
&= (n-L)j + \sigma_L^2 - \sigma_L(j-1) - \sigma_L \\
&= (n-L)j + \sigma_L^2 - \sigma_L(j-1) + \frac{1}{4}(j-1)^2 - \frac{1}{4}(j-1)^2 - \sigma_L \\
&= (n-L)j + (\sigma_L - \frac{1}{2}(j-1))^2 - \frac{1}{4}(j-1)^2 - \sigma_L \\
&\geq (n-L)j - \frac{1}{4}(j-1)^2 - \sigma_L \\
&\geq (n-L)j - \frac{1}{4}(j-1)^2 - (j-1). \qquad (10)
\end{aligned}
$$

So by relations (7)–(9), the value L satisfies

$$e \leq \binom{L}{2} + L(n-L) - \frac{1}{2}(n-L) + \frac{1}{2}(n-L)j. \qquad (11)$$

Also, by relations (7), (8), and (10), the value L satisfies

$$e > \binom{L-1}{2} + (L-1)(n-L+1) - \frac{1}{2}(n-L+1) + \frac{1}{2}(n-L+1)j - \frac{1}{8}(j-1)^2 - \frac{1}{2}(j-1). \qquad (12)$$

From relation (11), by finding the roots of the corresponding second-order (in L) equation, we have that L must satisfy

$$\frac{1}{2}\left(2n-j-\sqrt{(2n-j)^2+4n(j-1)-8e}\right) \leq L \text{ and} \qquad (13)$$

$$\frac{1}{2}\left(2n-j+\sqrt{(2n-j)^2+4n(j-1)-8e}\right) \geq L. \qquad (14)$$

Similarly, from relation (12), we have that either

$$L < \frac{1}{2}\left(2n-j+2-\sqrt{(2n-j)^2+4n(j-1)-8e+j^2-1}\right) \text{ or} \qquad (15)$$

$$L > \frac{1}{2}\left(2n-j+2+\sqrt{(2n-j)^2+4n(j-1)-8e+j^2-1}\right). \qquad (16)$$

Finally, from inequalities (13)–(16), we obtain that

$$
\begin{aligned}
L &\geq \frac{1}{2}\left(2n-j-\sqrt{(2n-j)^2+4n(j-1)-8e}\right) \text{ and} \\
L &< \frac{1}{2}\left(2n-j+2-\sqrt{(2n-j)^2+4n(j-1)-8e+j^2-1}\right).
\end{aligned}
$$

This concludes the proof of the theorem. □

The following corollary is now immediate.

COROLLARY 3.6. *For a graph G with n vertices and e edges,*

$$j\text{-}linkage(G) \geq \frac{1}{2}\left(2n-j-\sqrt{(2n-j)^2+4n(j-1)-8e}\right).$$

For the case $j = 1$, not only is the lower bound for the linkage of a graph given in this paper greater than or equal to the lower bound given by Erdös in [6] (this is an immediate conclusion of the fact that we give a tight lower bound) but, moreover, as the following proposition shows, when the number of edges exceeds a low threshold, the inequality is proper.

PROPOSITION 3.7. *Let* $L^{ER}(n, e) = \lceil e/n \rceil$. *Then* $L(e, n, 1)$ *is strictly greater than* $L^{ER}(e, n)$ *for any* $e \geq$ max-$E(n, 1, i-1) + 1$, *where* $i = \lceil (-1 + \sqrt{1+8n})/2 \rceil$.

Proof. Let $i = \lceil (-1 + \sqrt{1+8n})/2 \rceil$. Also, let e be such that $e \geq$ max-$E(n, 1, i-1) + 1$. First, observe that

$$\forall k \geq i, \quad \text{max-}E(n, 1, k) \leq (k-1)n. \tag{17}$$

Now recall (see relation (7)) that $L(e, n, 1)$ (L for short) is the unique integer for which

$$\text{max-}E(n, 1, L-1) < e \leq \text{max-}E(n, 1, L). \tag{18}$$

Also, max-$E(n, 1, k)$ is a strictly increasing function of k and therefore its values delimit on the real line a succession of consecutive intervals. By relation (18), max-$E(n, 1, L-1)$ and max-$E(n, 1, L)$ are the two endpoints of the one among these intervals that contains e.

On the other hand, it is trivial to see that $L^{ER}(e, n)$ (L^{ER} for short) is the unique integer for which

$$(L^{ER} - 1)n < e \leq L^{ER} n. \tag{19}$$

Also, kn is a strictly increasing function of k and therefore its values delimit on the real line a succesion of consecutive intervals. By relation (19), $(L^{ER} - 1)n$ and $L^{ER}n$ are the two endpoints of the one among these intervals that contains e.

By relation (17), for all $k \geq i$, the intervals delimited by the values of the function max-$E(n, 1, k)$ lie completely to the left of the intervals delimited by the corresponding values of the function kn. Finally, for any $e >$ max-$E(n, 1, i-1), L(e, n, 1) \geq i$. Therefore, we conclude that $\forall e >$ max-$E(n, 1, i-1)$, $L^{ER}(n, e) < L(e, n, 1)$ because otherwise, e would have to belong to two disjoint intervals. □

3.2. A lower bound for treewidth, pathwidth, and bandwidth.

In this subsection, we show that $L(e, n, 1)$, which is equal to $\frac{1}{2}(2n - 1 - \sqrt{(2n-1)^2 - 8e})$, is a *tight* lower bound for treewidth, pathwidth, and bandwidth. (Recall that $L(e, n, 1)$ is a tight lower bound for width.) A proof that $L(e, n, 1)$ is a lower bound for bandwidth can be found in [15]. In [20], X. Yan independently proved that $L(e, n, 1)$ is a lower bound for treewidth and pathwidth (see also [16]). However, in none of these papers is it shown that $L(e, n, 1)$ is a *tight* lower bound.

First, for completeness, we give some formal definitions (see also [4, 10, 17]).

DEFINITION 1. *A* tree-decomposition *of* $G = (V, E)$ *is defined to be a pair* $(\{X_i : i \in I\}, T)$, *where* $\{X_i : i \in I\}$ *is a collection of subsets of* V *and* $T = (I, F)$ *is a tree having the index set* I *as set of vertices, such that the following conditions are satisfied:*

1. $\bigcup_{i \in I} X_i = V$.
2. $\forall \{u, w\} \in E, \ \exists i \in I : u, w \in X_i$.
3. $\forall i, j, k \in I$: *if* j *is on a path in* T *from* i *to* k, *then* $X_i \cap X_k \subseteq X_j$.

The treewidth of a tree-decomposition $(\{X_i : i \in I\}, T)$ *is defined to be equal to* $\max_{i \in I} |X_i| - 1$. *The* treewidth of G *is defined to be the minimum treewidth of any tree-decomposition of* G.

DEFINITION 2. *A* path-decomposition *of* $G = (V, E)$ *is defined to be a class* $\{X_i : i = 1, \ldots, r\}$ *of subsets of* V *such that the following conditions are satisfied:*

1. $\bigcup_{i=1}^{r} X_i = V$.
2. $\forall \{u, w\} \in E,\ \exists i : u, w \in X_i$.
3. $\forall i, j, k$, *if* $1 \leq i \leq j \leq k \leq r$, *then* $X_i \cap X_k \subseteq X_j$.

The pathwidth of a path-decomposition $\{X_i : i = 1, \ldots, r\}$ *is defined to be equal to* $\max_{1 \leq i \leq r} |X_i| - 1$. *The* pathwidth of G *is defined to be the minimum pathwidth of any path-decomposition of* G.

DEFINITION 3. *The* bandwidth of a layout $l = (v_1, \ldots, v_n)$ *of* $G = (V, E)$ *is defined to be equal to* $\max\{|i - j| : \{v_i, v_j\} \in E\}$. *The* bandwidth of G, *is defined to be the minimum bandwidth of any layout of* G.

Treewidth has many equivalent characterizations (see, e.g., [2, 4]). We are going to use the one expressed in terms of elimination orderings of graphs [2]. An *elimination ordering* of a graph $G = (V, E)$ is an ordering $\pi = (v_1, \ldots, v_n)$ of the vertices of G. The graphs generated during an elimination of the vertices of G according to π are defined as follows: $G_1 = G$; G_{i+1} is equal to the graph obtained from G_i by deleting the vertex v_i and adding new edges (if necessary) so that all pairs of neighbors of v_i in G_i are adjacent in G_{i+1}. Obviously, G_{n+1} is equal to the empty graph. The *dimension of* v_i with respect to π is defined to be the degree of v_i in G_i. The *dimension of* π is the maximum dimension of any of the v_i's. Finally, the *elimination dimension* of G is the minimum dimension of any elimination ordering of G. The following result can be found in [2].

THEOREM 3.8 (Arnborg [2]). *The treewidth of a graph is equal to its elimination dimension.*

We now show the following.

THEOREM 3.9. *The quantity* $L(e, n, 1)$ *is a lower bound for the treewidth, pathwidth, and bandwidth of a graph with* e *vertices and* n *edges. Moreover, for arbitrary* n *and* e, *with* $n \geq 1$ *and* $0 \leq e \leq \binom{n}{2}$, *there is a graph* G *with* e *edges and* n *vertices such that its pathwidth, treewidth, and bandwidth are all equal to* $L(e, n, 1)$. *(i.e.,* $L(e, n, 1)$ *is a tight lower bound for the aforementioned parameters).*

Proof. From the characterization of treewidth in terms of elimination orderings, it is easy to see that linkage$(G) \leq$ treewidth(G). Also, obviously, treewidth$(G) \leq$ pathwidth (G). Finally, treewidth$(G) \leq$ bandwidth (G) (see [3]). Therefore, $L(e, n, 1)$ is a lower bound of all three parameters, as the theorem requires.

In order to prove that $L(e, n, 1)$ is also a tight lower bound, it suffices to construct, for given e and n, a graph G with e edges and n vertices such that pathwidth$(G) =$ treewidth$(G) =$ bandwidth$(G) = L(e, n, 1)$. First, we arrange n vertices on a layout $l = \{v_1, \ldots, v_n\}$ and connect two vertices v_i, v_j $(1 \leq i, j \leq n)$ with an edge iff $|i - j| \leq L - 1$ (this graph is described in [15]; see also [3, 17]). Call this graph H and note that it has $\binom{L-1}{2} + (n - (L - 1))(L - 1)$ edges. The graph G is constructed from H by adding to the latter the edges $\{v_1, v_{L+1}\}, \ldots, \{v_d, v_{L+d}\}$, where $d = e - (\binom{L-1}{2} + (n - (L - 1))(L - 1))$ (see Fig. 3). It is easy to see that the graph G has linkage, treewidth, pathwidth, and bandwidth all equal to $L(e, n, 1)$. This completes the proof of the theorem. □

Notice that for the cases of treewidth and pathwidth, the extremal graph construction of Theorem 3.4 for the case $j = 1$ is also sufficient to prove the tightness of

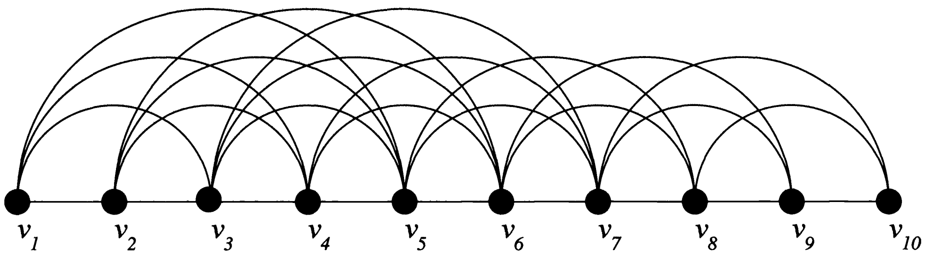


FIG. 3. *The construction of Theorem* 3.9 *for* $n = 10$, $e = 27$, *and* $L = 4$.

the lower bound.

3.3. Upper bound. In this subsection, we derive a tight upper bound for the j-linkage of graphs with given numbers of vertices and edges. In analogy to the previous subsection, we consider the graphs whose j-linkage decreases by the deletion of even one edge. We then consider the minimum number of edges that such a graph may have, given its j-linkage and its number of edges. To proceed, define the following function:

$$\text{min-}E(n, j, k) = \begin{cases} \lceil (k+j)k/2 \rceil & \text{if } 1 < k \leq n-j, \\ j & \text{if } k = 1, \\ 0 & \text{if } k = 0. \end{cases}$$

LEMMA 3.10. *let G be a graph with j-linkage at least k. Suppose that G has the property that any graph obtained from G by deleting one of its edges has j-linkage equal to $k-1$. Then the number of edges of G is at least* min-$E(n, j, k)$. *Moreover, for any $n \geq 1$, for any $j = 1, \ldots, n$ and any k such that $0 \leq k \leq n-j$, there exist graphs with n vertices,* min-$E(n, j, k)$ *edges, and j-linkage equal to k and such that if we subtract one edge, the j-linkage decreases to $k-1$.*

Proof. Assume first that $k > 1$. By hypothesis, we have the following:

1. j-linkage$(G) = k$;
2. j-linkage$(G') = k - 1$, where G' is any graph obtained from G by deleting one edge.

Property 1 above implies that the $(k-1)$-elimination cannot remove all vertices of G, and property 2 implies that the $(k-1)$-elimination removes all vertices of any graph G' obtained from G by removing an arbitrary edge.

Observe that the $(k-1)$-elimination applied to G not only returns a nonempty graph but, moreover, does not remove any nonisolated vertex of G (i.e., a vertex with degree in $G \geq 1$). Indeed, otherwise, the $(k-1)$-elimination would also remove at least one edge—say $\{v, u\}$—of G and therefore this $(k-1)$-elimination applied to $G - \{u, v\}$ would return a nonempty subgraph—a contradiction.

Let the number of nonisolated vertices in G be n'. Since $k \geq 1$, $n' > 0$. Observe that a nonisolated vertex must have at least k incident edges (otherwise, it would be removed by the $(k-1)$-elimination). Therefore, the number of edges of G is $\geq n'k/2$.

Now, observe that $n' \geq k + j$ because if $0 < n' < k + j$, then nonisolated vertices of G would be removed by the $(k-1)$-elimination.

Therefore, from the last two facts, we have that

$$\text{min-}E(n, j, k) = \lceil (k+j)k/2 \rceil$$

is a lower bound for e.

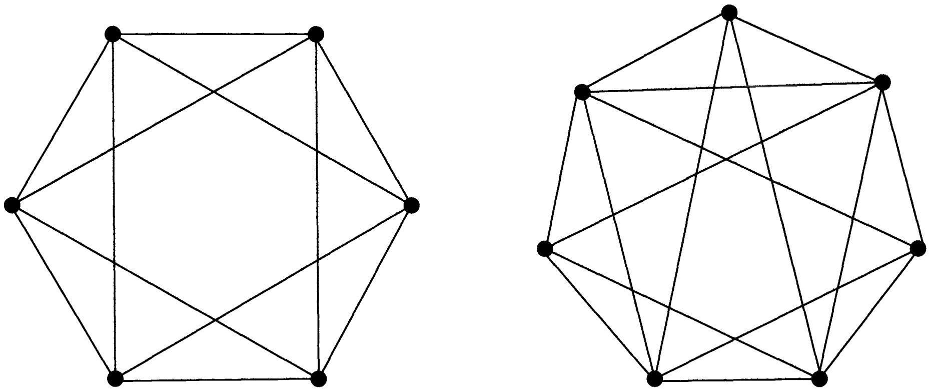


FIG. 4. *The construction of Lemma* 3.10 *for* $j = 2$, $k = 4$, *and* $k = 5$.

Finally, it is easy to see that for $k = 1$, a graph with the given properties should have at least $j = \text{min-}E(n, j, 1)$ edges; the case $k = 0$ is trivial since $\text{min-}E(n, j, 0) = 0$. This completes the prooof of the first statement of the lemma.

To show the second statement, given $n \geq 1$, $j = 1, \ldots n$, and any $2 \leq k \leq n - j$, we construct a graph with n vertices, $\text{min-}E(n, j, k)$ edges, and j-linkage equal to k and such that the j-linkage becomes $k - 1$ if one edge is deleted. For this construction, consider $k + j$ vertices with originally no edges connecting them. It is convenient to think of these vertices as cyclically lying on the plane. Connect with an edge any pair of these vertices whose cyclic distance is at most $\lfloor k/2 \rfloor$. Also, only in the case where k is odd, arbitrarily choose $\lceil (k + j)/2 \rceil$ successive vertices on the circle and connect each of them with the vertex at clockwise distance $\lfloor (k + j)/2 \rfloor$ (see Fig. 4). Finally, add to this graph $n - k - j$ isolated vertices. It is easy to see that this graph satisfies the requirements. For the case $k = 1$, consider an arbitrary tree with $j + 1$ vertices and add $n - j - 1$ isolated vertices. This graph again satisfies the requirements. For the case where $k = 0$, consider a graph with no edges. □

The following two lemmas can be easily proved analogously to Lemmas 3.2 and 3.3, respectively. We omit the proofs.

LEMMA 3.11. *A graph G with j-linkage$(G) \geq k$ has at least* $\text{min-}E(n, j, k)$ *edges.*

LEMMA 3.12. *The function* $\text{min-}E(n, j, k)$ *is strictly increasing with respect to k.*

Given natural numbers $n \geq 1$, $j = 1, \ldots, n$, and e such that $0 \leq e < \text{min-}E(n, j, n - j)$, let $U(e, n, j)$ (or just U when no confusion may arise) be the unique integer that satisfies the following relation:

$$\text{min-}E(n, j, U(e, n, j)) \leq e < \text{min-}E(n, j, U(e, n, j) + 1). \tag{20}$$

Notice that the uniqueness of U follows from Lemma 3.12. Moreover, for completeness, we set $U(e, n, j) = n - j$ for any $n \geq 1$, $j = 1, \ldots, n$, and e such that $\text{min-}E(n, j, n - j) \leq e \leq \binom{n}{2}$.

We now prove the following theorem.

THEOREM 3.13. *The function $U(e, n, j)$ is an upper bound for the j-linkage of graphs with n vertices and e edges. Moreover, for arbitrary n and e such that $n \geq 1$ and $0 \leq e \leq \binom{n}{2}$, there is a graph G with e edges and n vertices such that j-linkage$(G) = U(e, n, j)$ (i.e, $U(e, n, j)$ is a* tight *upper bound for the j-linkage).*

Proof. We give the proof only for $0 \leq e < \text{min-}E(n, j, n - j)$. The case where $\text{min-}E(n, j, n - j) \leq e \leq \binom{n}{2}$ is similar and easier. The fact that $U(e, n, j)$ is an upper bound follows easily from Lemma 3.11 and relation (20). In order to prove that $U(e,n,j)$ is a tight upper bound, if suffices to construct for given e, n, and j a graph with e edges, n vertices, and j-linkage equal to U. We distinguish the following three cases:

1. $\text{min-}E(n,j,0) = 0 \le e < j = \text{min-}E(n,j,1)$,
2. $\text{min-}E(n,j,1) = j \le e < j+2 = \text{min-}E(n,j,2)$,
3. $\text{min-}E(n,j,2) \le e < \text{min-}E(n,j,n-j)$.

For the first case, we have only to notice that for any graph G with n vertices and e edges, if $e < j$, then j-linkage$(G) = 0$; thus the construction for this case is obvious..

For the second case, we can see that any tree with $j+1$ vertices and j edges has j-linkage equal to 1. Also, any circle with $j+1$ vertices and $j+1$ edges has j-linkage equal to 1.

For the third case, by applying the second part of Lemma 3.10 for $k = U$, we obtain a graph—call it H—with $\text{min-}E(n,j,U)$ edges such that j-linkage$(H) = U$. We will describe a way to add $d = e - \text{min-}E(n,j,U)$ new edges to H so that the the resulting graph has the required properties. Observe that by relation (20), $0 \le d < \text{min-}E(n,j,U+1) - \text{min-}E(n,j,U)$. We first claim that also $d < U + j$. Indeed, we have the following:

1. If j is odd, then $\text{min-}E(n,j,U+0,n,j) - \text{min-}E(n,j,U) = U + \frac{j+1}{2}$.
2. If j is even, then
   - if U is odd, then $\text{min-}E(n,j,U+1) - \text{min-}E(n,j,U) = U + \frac{j}{2}$;
   - if U is even, then $\text{min-}E(n,j,U+1) - \text{min-}E(n,j,U) = U + 1 + \frac{j}{2}$.

From the analysis above, we observe that in any case,

$$\text{min-}E(n,j,U+1) - \text{min-}E(n,j,U) \le U + j.$$

Now, from its construction, we see that H has at least one isolated vertex and $U + j$ nonisolated ones. We construct the graph G by connecting the isolated vertex of H with d nonisolated vertices of it. This is possible since, as we showed above, $d < U + j$. It is now easy to see that the graph G has j-linkage equal to U. □

Since $\text{min-}E(n,j,U)$ is given by a closed formula, $U(e,n,j)$ can be efficiently computed. Moreover, as the following theorem shows, we can find an almost closed formula for computing $U(e,n,j)$.

THEOREM 3.14. *For any* $n \ge 1$, $j = 1,\dots,n$, $j+2 \le e \le \binom{n}{2}$,

$$U(e,n,j) > \frac{1}{2}\left(-j + \sqrt{j^2 - 4 + 8e}\right) - 1 \quad \textit{and}$$
$$U(e,n,j) \le \frac{1}{2}\left(-j + \sqrt{j^2 + 8e}\right).$$

Proof. For notational convenience, we define

$$\delta_j = (U+j)U \bmod 2.$$

We first see that for $e \ge j+2$,

$$\text{(21)}\qquad \text{min-}E(n,j,U) = \lceil (U+j)U/2 \rceil = \begin{cases} \frac{1}{2}(U+j)U & \text{if } \delta_j = 0, \\ \frac{1}{2}((U+j)U+1) & \text{if } \delta_j = 1. \end{cases}$$

We distinguish the following two cases:

1. j is odd. Then $\delta_j = 0$.
2. j is even. Then
   - $\delta_j = 1$, if U is odd;
   - $\delta_j = 0$, if U is even.

For the first case, we have from relations (20) and (21) that

$$\frac{1}{2}(U+j)U \leq e < \frac{1}{2}(U+j+1)(U+1). \tag{22}$$

Now, from the inequalities in (22), by computing the roots of the second-order (in U) equations, we obtain that

$$U(e,n,j) = \left\lfloor \frac{1}{2}\left(-j+\sqrt{j^2+8e}\right)\right\rfloor. \tag{23}$$

For the second case, again by relations (20) and (21), we have that

$$\frac{1}{2}(U+j)U \leq e < \frac{1}{2}((U+j+1)(U+1)+1). \tag{24}$$

Now, as in Theorem 3.14, computing the roots corresponding the second-order equations (in U), we obtain that inequality (24) is satisfied for

$$U(e,n,j) > \frac{1}{2}\left(-j+\sqrt{j^2-4+8e}\right)-1 \text{ and} \tag{25}$$

$$U(e,n,j) \leq \frac{1}{2}\left(-j+\sqrt{j^2+8e}\right). \tag{26}$$

From relations (23), (25), and (26), we conclude that in all cases the requirement holds. □

The following corollary is now immediate.

COROLLARY 3.15. *For a graph G with n vertices and e edges,*

$$j\text{-}linkage(G) \leq \frac{1}{2}\left(-j+\sqrt{j^2+8e}\right).$$

By easier calculations, it can be proved that $\min(2e/j, (2e)^{1/2})$ is also an upper bound for the j-linkage. However, when $j = \Theta((2e)^{1/2})$, the difference of this bound from the optimal one that we provide is $\Theta(e^{1/2})$.

CONCLUSION. *From the above results, we conclude that for a graph G with n vertices and $e \geq j+2$ edges, an estimation for the j-linkage(G)(equivalently, the j-width(G)) is given by the following two inequalities:*

$$\left\lceil \frac{1}{2}\left(2n-j-\sqrt{(2n-j)^2+4n(j-1)-8e}\right)\right\rceil \leq j\text{-}linkage(G),$$

$$\left\lfloor \frac{1}{2}\left(-j+\sqrt{j^2+8e}\right)\right\rfloor \geq j\text{-}linkage(G).$$

The estimation from below is tight within an additive term $O(j)$, whereas the estimation from above is tight within an additive constant. (If $e \leq j+1$, j-linkage(G) is equal to 0 or 1.)

4. Parallel complexity and approximations. The problem of computing of the j-linkage of a graph G can be formulated as a decision problem: Given a graph G and an integer k, determine whether $G^{(k,j)}$ (the graph that the (k,j)-Elimination Algorithm outputs on input G) is not empty. A stronger decision problem is: Given a graph G, an integer $k \geq 0$ and a vertex u of G, determine whether $u \in G^{(k,j)}$. For $j = 1$, these problems are examined by Anderson and Mayr in [1]. However not all their proofs generalize to arbitrary j. A related problem, which exhibits a threshold-type complexity, can be found in [9] (see also [18]).

4.1. An NC algorithm and P-completeness.

THEOREM 4.1. *The problem of deciding whether $u \in G^{(1,j)}$ is in* NC *(with j as part of the input).*

Proof. We give an algorithm in NC that on input G returns $G^{(1,j)}$:

Find all biconnected components of G. It is easy to see that all vertices of such a component with strictly more than $j+1$ vertices are contained in $G^{(1,j)}$. So we have to face the problem of which vertices in biconnected components with $\leq j+1$ vertices remain in $G^{(1,j)}$. To solve this problem, construct a new graph H that has three groups of vertices: (i) one vertex for each biconnected component of G with at most $j+1$ vertices, (ii) one vertex for each articulation point of G that belongs only to biconnected components of vertex cardinality at most $j+1$, and (iii) as many vertices as the vertices of G that belong to biconnected components with strictly more than $j+1$ vertex cardinality. Now, connect two vertices in group (iii) iff the they are connected in G as well. Also connect a vertex in group (i) with a vertex in group (iii) (respectively, group (ii)) iff the vertex of group (iii) (respectively, (ii)) corresponds to an articulation point of the biconnected component represented by the vertex of group (i).

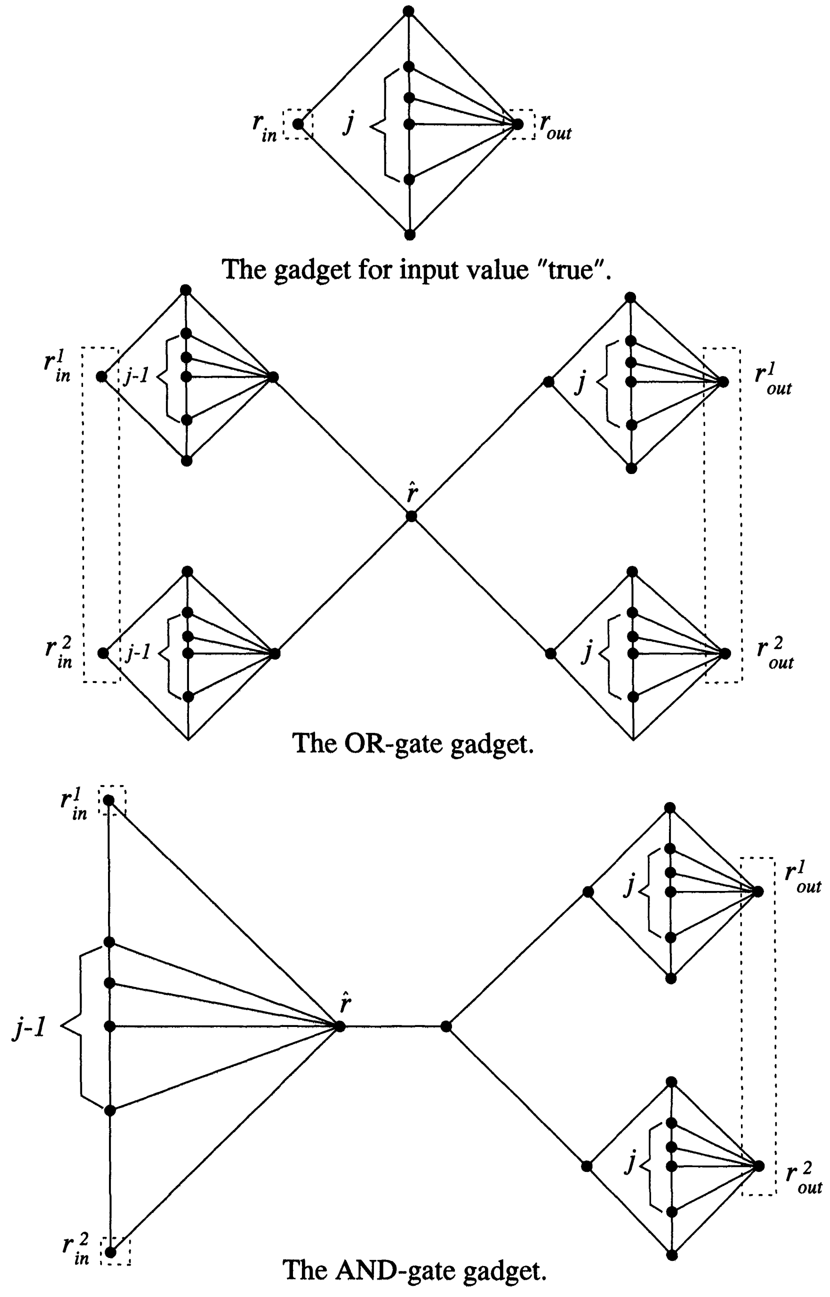
In the graph thus constructed, recursively delete all vertices that are elements of chains (a chain is defined to be a path of vertices that starts with a vertex of degree 1 and contains no vertex of degree strictly greater than 2). The recursion is repeated until no chains appear in the current graph. Then reconstruct the part of G that corresponds to the part of H that has remained. Since the creation of a new vertex of degree 1 in the current H requires the removal of at least two chains, the number of chains removed decreases by at least half at each phase of the recursion. Therefore, it is not hard to see that this is an algorithm in NC that outputs $G^{(1,j)}$. □

THEOREM 4.2. *If j and $k > 1$ are fixed, it is* P*-complete for a graph G to determine whether a given vertex u lies in $G^{(k,j)}$ or even to determine whether $G^{(k,j)}$ is nonempty.*

Proof. The proof of this result is a straightforward generalization of the corresponding result in [1], where the P-completeness for $j = 1$ is proved by a reduction from the fanout-2 monotone circuit value problem. Therefore, we omit the proof; however, for the sake of completeness, in Fig. 5 we give the modified gadgets for arbitrary j (for the problem of determining whether $G^{(k,j)}$ is nonempty). □

4.2. Parallel approximations: A threshold behavior. In this subsection, we consider the problem of approximating the j-linkage of a graph G. An algorithm with approximation factor a constant $c > 0$ is one that on input G returns an integer $j\text{-linkage}_{\mathrm{ap}}(G)$ such that

$$j\text{-linkage}(G) \geq j\text{-linkage}_{\mathrm{ap}}(G) \geq c(j\text{-linkage}(G)).$$



The gadget for input value "true".

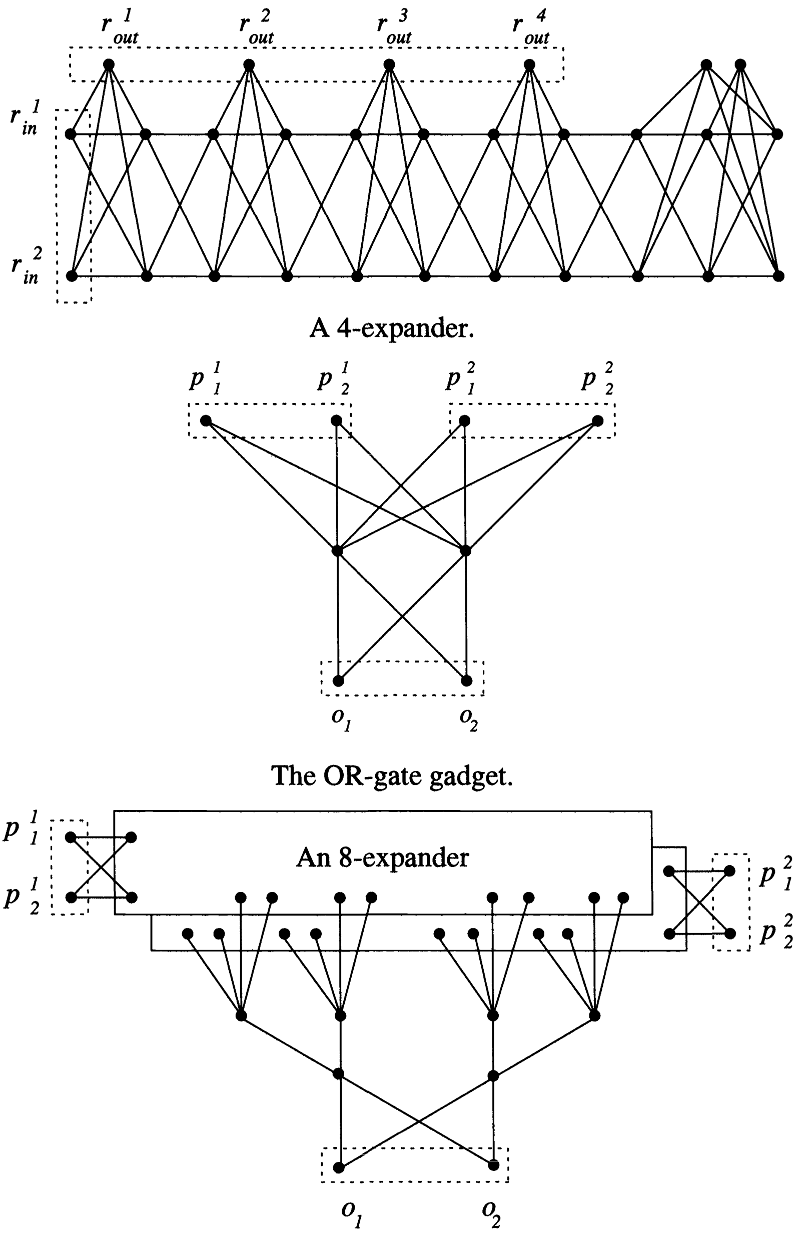
The OR-gate gadget.

The AND-gate gadget.

FIG. 5. *The gadgets of Theorem* 4.2.

THEOREM 4.3. *For any constant* $0 < c < 1/(2j)$, *for fixed* j, *the problem of approximating the* j-*linkage of a graph* G *with an approximation factor equal to* c *is in NC.*

Proof. First, observe that $\lceil e/(nj) \rceil \leq L(e, n, j)$. Following Anderson and Mayr in [1], we define a procedure Test(k) that returns either that the graph has no subgraph of j-min-degree at least k or that the graph has j-linkage at least $\frac{1-\epsilon}{2j}k$. The procedure



A 4-expander.

The OR-gate gadget.

The AND-gate gadget.

FIG. 6. *The gadgets of Theorem* 4.4.

deletes in parallel all sets of vertices S whose cardinality is at most j and whose external degree with respect to the current graph G' is strictly less than k until the ratio of the vertices of G' that are contained in at least one such set S becomes $\leq \epsilon$ (ϵ is to be determined). This procedure is in NC. If it returns the empty graph, then G has no subgraph of j-min-degree at least k. If the procedure returns a subgraph G' with $n' > 0$ vertices, then the number of vertices with degree strictly less than k is at most

$\epsilon n'$, so G' has at least $\frac{(1-\epsilon)n'k}{2}$ edges, and so it follows that j-linkage$(G) \geq \lceil \frac{1-\epsilon}{2j} k \rceil$.

We then apply the procedure Test(k) in parallel for $k = 0, \ldots, n$, and we thus find a value k_0 such that G has no subgraph with j-min-degree $> k_0$ but it has a j-linkage at least $\lceil \frac{1-\epsilon}{2j} k_0 \rceil$. So the j-linkage of the graph will be a value between $\lceil \frac{1-\epsilon}{2j} k_0 \rceil$ and k_0. The algorithm finally returns $\lceil \frac{1-\epsilon}{2j} k_0 \rceil$ as an aproximation value for the j-linkage. Given any $c < \frac{1}{2j}$, choose a suitable ϵ so that the value returned as an approximation satisfies

$$j\text{-linkage}(G) \geq j\text{-linkage}_{\text{ap}}(G) \geq c(j\text{-linkage}(G)).$$

This completes the proof. □

THEOREM 4.4. *If* P $\neq$ NC, *then for any fixed j, it is not possible to approximate the j-linkage(G) by a factor strictly greater than $\frac{1}{2}$ in* NC.

Proof. We follow the methodology used in the proof of the case where $j = 1$. We give a logspace transformation of an instance of the monotone circuit value problem to a graph G such that G has j-linkage equal to $k+1$ if the output of the circuit is "false", whereas it has j-linkage equal to $2k$ if the output of the circuit is "true".

For the construction of the gadgets that simulate the circuit, we need an expander that propagates the values of the circuit. This expander is shown in Fig. 6. The k leftmost vertices $r_{in}^1, r_{in}^2, \ldots, r_{in}^k$ of this gadget are called the in-vertices of the expander (in Fig. 6, $k = 2$). The m top vertices $r_{out}^1, r_{out}^2, \ldots, r_{out}^m$ are called the out-vertices of the expander (we call such an expander an m-expander; m is chosen as needed; for the first expander in Fig. 6, $m = 4$). Observe that for large enough m, any set $S, |S| \leq j$, of vertices in an m-expander has external degree strictly more than k.

The gadgets for OR-gates and AND-gates of the circuit are given in Fig. 6. Observe that in the AND-gate gadget, we use two copies of a k^3 expander. The two clusters of in-vertices of these expanders are the two clusters of vertices corresponding to the input of the AND-gate. The vertices corresponding to the output of the AND-gate are connected as in Fig. 6 with the out-vertices of the expanders. We then identify the output clusters of each gate with the corresponding, according to the circuit structure, input clusters of other gates.

Let now r be the number of clusters of vertices of G that correspond to gates of the circuit that are not connected to previous gates, but originally have the value "true" as input. We identify these r clusters of vertices of G (containing k vertices each) with the kr out-vertices of a kr-expander. The in-vertices of this kr-expander are identified with the vertices corresponding to the final output gate of the circuit. This completes the construction of G.

Now it can be proved that if the circuit outputs the value "true", then j-linkage(G) $= 2k$ and if the circuit outputs the value "false", then j-linkage$(G) = k+1$. Indeed, if the circuit outputs the value "true", then it is easy to see that a $(2k, j)$-elimination removes all vertices of G, whereas a $(2k-1, j)$-elimination leaves a nonempty subgraph; moreover, if the circuit outputs the value "false", then a $(k+1, j)$-elimination removes the vertices of G, whereas a (k, j)-elimination leaves a nonempty subgraph (because the expander construction has a subgraph with j-min-degree equal to $k+1$). □

Acknowledgments. We thank the anonymous reviewers of previous drafts of this paper for their comments, which helped in improving the presentation and in correcting errors. In particular, we acknowledge a suggestion by a referee that simplified the proof of of Lemma 3.1. We also thank Paul G. Spirakis. He showed

us that—contrary to a common assumption—approximations need not be inherently iterative. They may well be done in parallel.

REFERENCES

[1] R. Anderson and E. Mayr, *Parallelism and greedy algorithms*, Adv. Comput. Res., 4 (1987) pp. 17–38; see also *A* P-*complete problem and approximations to it,* Tech. Report, Department of Computer Science, Stanford University, Stanford, CA, 1984.

[2] S. Arnborg, *Efficient algorithms for combinatorial problems on graphs with bounded decomposability: A survey*, BIT, 25 (1985), pp. 2–33.

[3] H. L. Bodlaender, *Classes of graphs with bounded treewidth*, Tech. Report RUU-CS-86-22, Department of Computer Science, Utrecht University, Utrecht, The Netherlands, 1986.

[4] ———, *A tourist guide through treewidth*, Acta Cybernet. 11 (1993), pp. 1–23.

[5] R. Dechter, *Directional resolution: The Davis–Putnam procedure, revised*, working notes, AAAI Spring Symposioum on AI and NP-Hard Problems, Stanford University, Stanford, CA, 1993, pp. 29–35.

[6] P. Erdös, *On the structure of linear graphs*, Israel J. Math., 1 (1963), pp. 156–160.

[7] E. C. Freuder, *A sufficient condition for backtrack-free search*, J. Assoc. Comput. Mach., 29 (1982), pp. 24–32.

[8] ———, *A sufficient condition for backtrack-bounded search*, J. Assoc. Comput. Mach., 32 (1985), pp. 755–761.

[9] L. Kirousis, M. Serna, and P. Spirakis, *The parallel complexity of the subgraph connectivity problem*, SIAM J. Comput., 22 (1993), pp. 573–586.

[10] J. van Leeuwen, *Graph algorithms*, in Handbook of Theoretical Computer Science, J. van Leeuwen, ed., Elsevier, New York, 1990, pp. 525–631.

[11] D. R. Lick and A. T. White, *k-Degenerate graphs*, Canad. J. Math., 22 (1970), pp. 1082–1096.

[12] D. W. Matula, *A min-max theorem for graphs with application to graph coloring*, SIAM Review, 10 (1968), pp. 481–482.

[13] A. Mackworth, *Constraint satisfaction*, in Encyclopedia of Artificial Inteligence, S. C. Shapiro, ed., John Wiley, New York, 1992, pp. 276–285.

[14] D. W. Matula, G. Marble, and J. D. Isaacson, *Graph coloring algorithms*, in Graph Theory and Computing, Academic Press, New York, 1972, pp. 109–122.

[15] Z. Miller, *Graph Layouts*, in Applications of Discrete Mathematics, J. G. Michaels and K. H. Rozen, eds., McGraw–Hill, New York, 1991, pp. 365–393.

[16] S. Ramachandramurthi, *A lower bound for treewidth and its consequences*, in Proc. 20th International Workshop on Graph-Theoretic Concepts in Computer Science, Springer-Verlag, Berlin, New York, Heidelberg, 1994, pp. 14–25.

[17] J. B. Saxe, *Dynamic programming algorithms for recognizing small-bandwidth graphs in polynomial time*, SIAM J. Algebraic Discrete Meth., 1 (1980), pp. 363–369.

[18] M. Serna and P. Spirakis, *The approximability of problems complete for P*, in Proc. International Symposium on Optimal Algorithms, Springer-Verlag, Berlin, New York, Heidelberg, 1989, pp. 193–204.

[19] G. Szekeres and H. S. Wilf, *An inequality for the chromatic number of a graph*, J. Combin. Theory, 4 (1968), pp. 1–3.

[20] X. Yan, *A relative approximation algorithm for computing the pathwidth*, Master's thesis, Department of Computer Science, Washington State University, Pullman, WA, 1989.

SIAM J. COMPUT.
Vol. 25, No. 3, pp. 648–662, June 1996

008

AN ALGORITHM FOR LOCATING NONOVERLAPPING REGIONS OF MAXIMUM ALIGNMENT SCORE*

SAMPATH K. KANNAN† AND EUGENE W. MYERS‡

Abstract. In this paper, we present an $O(N^2 \log^2 N)$ algorithm for finding the two nonoverlapping substrings of a given string of length N which have the highest-scoring alignment between them. This significantly improves the previously best-known bound of $O(N^3)$ for the worst-case complexity of this problem. One of the central ideas in the design of this algorithm is that of partitioning a matrix into pieces in such a way that all submatrices of interest for this problem can be put together as the union of very few of these pieces. Other ideas include the use of candidate lists, an application of the ideas of Apostolico et al. [*SIAM J. Comput.*, 19 (1990), pp. 968–988] to our problem domain, and divide-and-conquer techniques.

Key words. sequence alignment, efficient algorithms, repeated regions

AMS subject classifications. 68P05, 68Q20, 68Q25, 92B08, 92B99

1. Introduction.

Let $A = a_1a_2...a_N$ be a sequence of length N, and let $A[p..q]$ denote the substring $a_pa_{p+1}...a_q$ of A. The problem we consider is that of finding the score of the best alignment between two substrings $A[p..q]$ and $A[r..s]$ under the the generalized Levenshtein model of alignment [6, 11], which permits substitutions, insertions, and deletions of arbitrary score. This problem is a formalization of the problem, encountered by molecular biologists, of automatically detecting repeated regions in DNA and protein sequences. This problem has recently been considered by Miller [8]. When there is no restriction that the regions be nonoverlapping, he points out that the problem can be solved in $O(N^2)$ time by a straightforward modification of the algorithm of Smith and Waterman [10] that finds the highest-scoring local alignment between two sequences. Miller then goes on to consider the restriction that the regions be nonoverlapping and presents a worst-case $O(N^3)$ algorithm which runs in $O(N^2)$ in practice. His method involves the use of the candidate-list paradigm, which we review briefly in §2 since we use some of these ideas in our algorithm. Miller calls nonoverlapping regions *twins*, and as his result indicates, this constraint appears to make the problem much harder. For the rest of the paper, we consider only the problem of finding the best-scoring twins.

There is a related problem where the goal is to find nonoverlapping regions which are *exact* repeats. This problem has been dealt with before and turns out to be of significantly lower complexity than the problem of finding the best-scoring twins. When the exactly repeating regions are required to be adjacent or *tandem*, i.e., when the goal is to find the longest substring of A of the form ww, Main and Lorentz [7] provide an $O(N \log N)$ algorithm. If gaps are allowed between exactly repeating substrings, the problem becomes even simpler and can be handled in $O(N)$ time by first creating a suffix tree and then computing the largest and smallest indices (of suffixes) which go through each internal node in post order. The path to the deepest

* Received by the editors February 4, 1994; accepted for publication (in revised form) November 8, 1994.

† Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104 (kannan@central.cis.upenn.edu). The research of this author was supported in part by NSF grant CCR-9108969.

‡ Department of Computer Science, University of Arizona, Tucson, AZ 85721 (gene@cs.arizona.edu). The research of this author was supported in part by NLM grant LM-04960 and NSF grant CCR-9002351.

internal node whose smallest and largest index suffixes are sufficiently far apart gives us the desired repeated substring.

In a development parallel to ours, Landau and Schmidt [5] have extended the algorithm of Main and Lorentz to find approximate *tandem* repeats with *K-or-less differences*, a thresholded variation of the problem considered here restricted to the simple Levenshtein measure of similarity (i.e., unit-cost insertion, deletion, and substitution). Under these stronger conditions, they were able to develop an $O(NK\log N\log K)$ threshold-sensitive algorithm. It is interesting to note that their algorithm provides a bridge between the $O(N\log N)$ exact algorithm of Main and Lorentz (where $K = 0$) and our unthresholded $O(N^2\log^2N)$ algorithm (where $K = N$) for generalized Levenshtein measures. While the results for exact matching might suggest that approximate tandem repeats would be harder to find than twins, we will show that our algorithm can easily be modified to report only tandem repeats as a special case.

The rest of the paper is organized as follows. In §2, we define some concepts and review the results that are used in the construction of our algorithms. In §3, we present a relatively simple algorithm for the problem of finding twins which runs in time $O(N^{2.5}\log^{0.5} N)$. This algorithm already incorporates some of the ideas used in the more complex algorithm and so serves as a useful preliminary exercise. In §4, we present an algorithm which achieves a running time of $O(N^2\log^2 N)$ which is within a polylog factor of being optimal. In §5, we consider the problem of finding the best twins under the condition that the best twins are of size no more than $O(N^{1-\epsilon})$ for arbitrarily small values of ϵ. For this problem, we present an algorithm which runs in time $O(N^2)$. In §6, we describe open problems mainly concerned with improving the space complexity of our algorithm.

2. Preliminaries. Throughout the paper, we wish to think about the problem in terms of finding paths in a weighted *edit graph* [9] and performing the computation over the associated *dynamic programming matrix* [11]. The edit graph for sequence A versus itself consists of a lower-triangular matrix of vertices (i, j) for $0 \leq j \leq i \leq N$ with up to three edges directed into (i, j): a *substitution* edge from $(i-1, j-1)$ weighted $\delta(a_i, a_j)$; an *insertion* edge from $(i-1, j)$ weighted $\delta(a_i, \epsilon)$; and a *deletion* edge from $(i, j-1)$ weighted $\delta(\epsilon, a_j)$. Edges from nonexistent vertices and substitution edges on the main diagonal are not present. The scoring scheme δ may be chosen arbitrarily but in most application contexts is such that edge weights are negatively biased and only the substitution of similar symbols are given positive score. As illustrated in Figure 1, any path from vertex (p, r) to (q, s) models an alignment between $A[p+1..q]$ and $A[r+1..s]$, and the weight of the path is the score of the alignment. The correspondence is isomorphic, and so it suffices to think in terms of finding high-scoring paths in the edit graph. Limiting the graph to the lower-triangular part simply eliminates local alignments that cannot be twins, because any path crossing the main diagonal aligns overlapping regions.

The Smith–Waterman algorithm for local alignments applied to the edit graph for A reduces to evaluating the following fundamental recurrence for $C(i, j)$, the cost of the best path to (i, j) from some predecessor in the graph, in lexicographical order of i and j:

$$\text{For } j \leq i : C(i, j) = \max\begin{cases} C(i-1, j-1) + \delta(a_i, a_j) & \text{if } i > j > 0, \\ C(i-1, j) + \delta(a_i, \epsilon) & \text{if } i > 0, \\ C(i, j-1) + \delta(\epsilon, a_j) & \text{if } j > 0, \\ 0 & \text{always.} \end{cases}$$

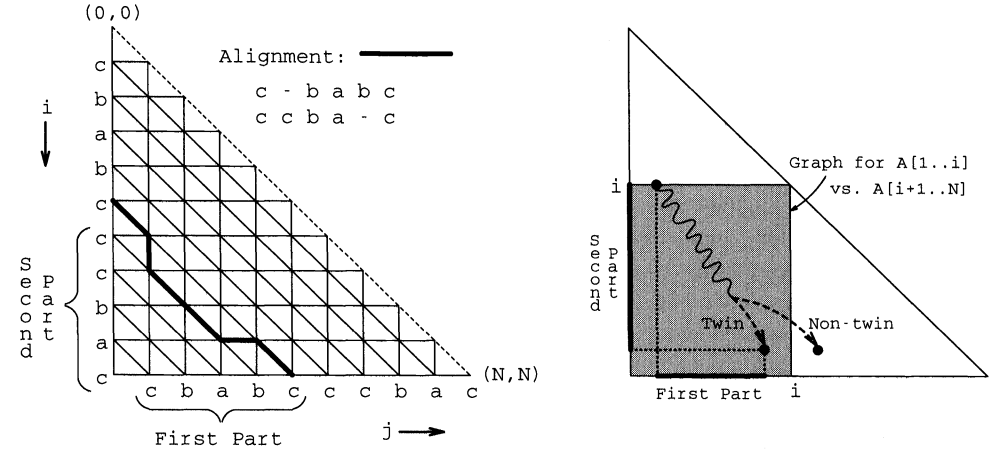


FIG. 1. *Edit-graph illustrations.*

The terms qualified by an *if* clause are present only if the condition is true. The score of the best substring alignment is given by $\max_{j \le i} \{C(i,\, j)\}$. Because the edit graph on A involves just the lower-triangular part of the underlying dynamic programming matrix, the trivial answer of aligning A with itself is precluded.

However, while the above finds the best-scoring path in the edit graph, it does not necessarily align nonoverlapping substrings. Figure 1 illustrates that if a path starts at $(i,\, j)$ and ends at $(x,\, y)$, then it models a twin only if $y \le i$, i.e., it ends in a column whose index is not greater than that of the row it starts in. Alternatively, a path is a twin if there exists an i such that the path lies entirely in the rectangular subgraph delimited by row i and column i, in which case we say the twins are *separated by* i. This observation leads to the obvious $O(N^3)$ algorithm for the twins problem: For each $i \in [1,\, N-1]$, run the Smith–Waterman algorithm for $A[1..i]$ versus $A[i+1..N]$, and record the best answer over all possible separators i.

Miller [8] obtained an algorithm for finding twins that is more efficient in practice by computing $C(i,\, j,\, k)$, the best path to $(i,\, j)$ from row k for each value of $k \le i$.

$$\text{For } j \le k \le i: \quad C(i,\, j,\, k) = \max \begin{cases} C(i-1,\, j-1,\, k) + \delta(a_i,\, a_j) & \text{if } i > j > 0 \text{ and } i > k, \\ C(i-1,\, j,\, k) + \delta(a_i,\, \epsilon) & \text{if } i > 0 \text{ and } i > k, \\ C(i,\, j-1,\, k) + \delta(\epsilon,\, a_j) & \text{if } j > 0 \text{ and } i \ge k, \\ 0 & \text{if } i = k. \end{cases}$$

Given these quantities, the best twin ending at $(i,\, j)$ is simply $\max_{j \le k} C(i,\, j,\, k)$. Of course, computing the above directly still takes $O(N^3)$ time, but Miller made the further observation that if $C(i,\, j,\, k) \ge C(i,\, j,\, h)$ and $k \ge h$, then there is no need to compute $C(i,\, j,\, h)$ because the other term can participate in any maximum twin that the former does. We say that k *dominates* h at $(i,\, j)$. Miller levers this observation by keeping at $(i,\, j)$ a list of *candidates* $(k,\, c)$ in increasing order of k such that $c = C(i,\, j,\, k)$ and k is not dominated by any other row at $(i,\, j)$ (which implies that the list is also in decreasing order of c). In practice, the number of rows not dominated at a vertex appears to be constant, and when this is true, computing a candidate list from the candidate lists of its immediate predecessors takes constant time. Thus Miller observes $O(N^2)$ behavior in practice, although it is possible that each candidate list could have as many as $\Omega(N)$ elements in it, and so take $O(N^3)$ in the worst case.

Note that finding tandem or adjacent twins is simply a matter of examining just the entries $C(i, j, j)$ (i.e., the best tandem twin ending at (i, j)) in the formulation above. So Miller's algorithm immediately solves the tandem variation as a special case. Since our algorithms effectively perform the same computation as Miller's, it will follow that both our simple and penultimate algorithms solve the tandem variation within the same complexities as they solve the basic twins problem.

We also make extensive use of the results of Apostolico et al. [1]. Although their paper is concerned with the design of a parallel algorithm for string matching, some of the techniques in it carry over to the sequential domain. Specifically, for any $m \times n$ rectangular[1] subgraph E of an edit graph where $m \leq n$, Apostolico et al. [1] show that the problem of computing the shortest distances between every one of the $n+m+1$ vertices on the left or top boundary of E to every one of the $n+m+1$ vertices on the right or bottom boundary of E can be done in $O((m+n)^2 \log n)$ time. We will refer to the resulting $(n+m+1) \times (n+m+1)$ table of distances between pairs of boundary vertices of E by $DIST_E$. Another result from this paper will also be relevant to us. This is an "incremental" version of the previous result and states that given a square, $m \times m$ edit graph E that is decomposed into four $m/2 \times m/2$ subgraphs A, B, C, and D by bisecting vertical and horizontal lines, $DIST_E$ can be computed from $DIST_A$, $DIST_B$, $DIST_C$, and $DIST_D$ in $O(m^2)$ time. Unfortunately, it is beyond the scope of this paper to explain the ingenious algorithms of Apostolico et al. that are based on a two-tiered development. However, we note that the procedure *Propagate* described later in this paper is essentially a small variation on the first tier of their design. We highly recommend that interested readers refer to this fundamental result.

3. A simple $O(N^{2.5}\log^{0.5} N)$ algorithm. The candidate-list technique of Miller does not give us a better-than-$O(N^3)$ algorithm because each list can get as large as $\Omega(N)$. In this section, we present an algorithm which achieves a better worst-case running time by eliminating the need for candidate lists with more than b elements in them, where we will choose b as a function of N later. Consider partitioning the interval $[0, N]$ into N/b *panels*, $[0, b], [b, 2b], \ldots, [N-b, N]$, with the *partition indices* $b, 2b, 3b, \ldots, N-b$. Consider the following first phase:

```
1. For each partition index $i = b, 2b, \ldots, N-b$, run the Smith–Waterman algorithm on $A[1..i]$ versus $A[i+1..N]$.
```

The step above detects every twin that is separated by one of the partition indices. Thus the only twins not captured are those where the first part ends and the second part begins within one of the panels. The second phase of our algorithm processes each panel in search of such *panel twins.* The outer loop of the second phase is as follows:

```
2. For each panel $[j, i] = [0, b], [b, 2b], \ldots, [N-b, N]$, do the following four steps:
```

Consider a panel twin of $[j, i]$. Figure 2 shows the path of such a twin which must begin at a vertex between rows j and i and which must end at a vertex between

[1] Here m and n refer to the length of the sides and not the number of vertices in them, which are $m+1$ and $n+1$, respectively.

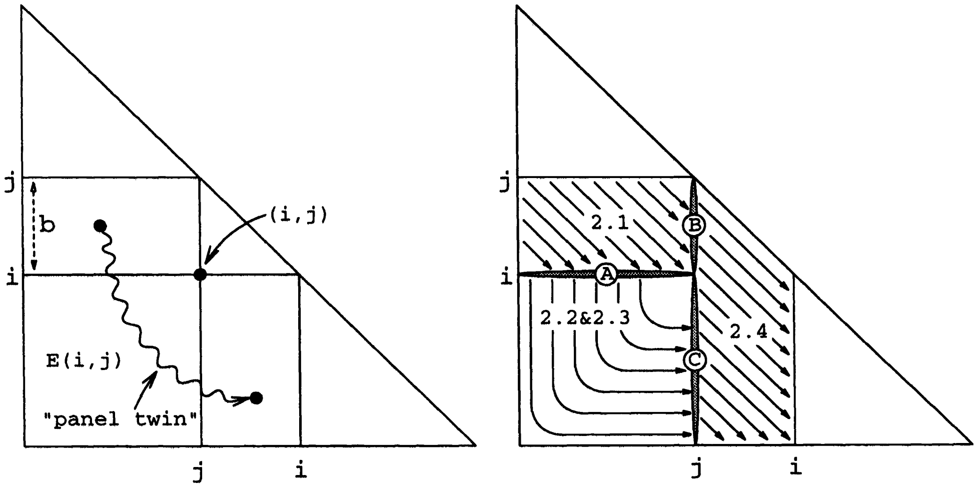


FIG. 2. *Panel processing for the simple algorithm.*

columns j and i. Thus it suffices to compute for all vertices (x, y) between columns j and i, the best paths originating from vertices between rows y and $\min(i, x)$. That is, $\max_{y \le z \le \min(i,x)} C(x, y, z)$ is the cost of the best panel twin ending at (x, y). Moreover, the maximum involves at most $b+1$ candidates, one from each row between j and i. To begin the processing of the panel, consider the following:

2.1. For each vertex on boundaries A and B shown in Figure 2, find the best paths to them that originate in rows j through i using Miller's recurrence.

Rigorously stated, this step computes $C(x, y, z)$ for $x \in [j, i]$, $y \in [0, j]$, and $z \in [j, x]$ in lexicographical order and retains the computed quantities at vertices (x, y) such that $x = i$ or $y = j$ in candidate *arrays* indexed by z. Now the crux of the problem is to compute the best paths originating in rows j through i to the vertices on boundary C shown in Figure 2. Using Miller's recurrence and computing all the necessary intermediate quantities in the rectangular subgraph, $E(i, j)$, whose upper right corner is vertex (i, j), would require $O(N^2b)$ time, which is too costly. This is circumvented by the next two steps as follows:

2.2. Compute $M = DIST_{E(i,j)}$ with the algorithm of Apostolico et al.

The table M of distances between pairs of points bounding $E(i, j)$ is used to efficiently "propagate" the candidate arrays on boundary A to boundary C. The basic observation is that $C(x, j, z) = \max_{0 \le y \le j} C(i, y, z) + M[(i, y)][(x, j)]$ for all $x \in [i, N]$ and $z \in [j, i]$. That is, a best path from row z to (x, j) must pass through row i and so is decomposable into (1) a best path to some vertex (i, y) on this row, followed by (2) a best path from (i, y) to (x, j). With this preliminary, the next step is as follows:

2.3. For each vertex on boundary C shown in Figure 2, efficiently compute the best paths originating in rows j through i by propagating the candidate lists from boundary A by consulting table M.

If the maximum embodying propagation were computed directly for each boundary-

C vertex, we would again take too much time. Fortunately, for a fixed originating row z, it is possible to compute $C(x, j, z)$ for all (x, j) on boundary C in $O(\log N)$ amortized time per vertex by the procedure *Propagate* given below.

Note that the pseudocode below assumes pass-by-reference semantics and that the index ranges of *Propagate*'s formal arguments are correlated with the matrix slices passed to it as actual arguments. Further note that $C(i, 0..j, z)$ is passed to $I[1..n]$ but that $C(N..i, j, z)$ is passed to $O[1..m]$. The indices must decrease in the later actual argument because *Propagate*'s divide-and-conquer strategy implicitly requires that the vertices on the boundary of $E(i, j)$ be ordered so that paths from the input boundaries (upper and left) to the output boundaries (lower and right) cross when their start and finish points are inverted with respect to this order. As the increasing input-boundary order, we use up the left side and then across the top to the right, and as the increasing output-boundary order, we use across the bottom to the right and then up the right side. For example, for $E(i, j)$ the increasing input-boundary order is $(N, 0), (N-1, 0), \ldots, (i+1, 0), (i, 0), (i, 1), \ldots, (i, j-1), (i, j)$, and the increasing output-boundary order is $(N, 0), (N, 1), \ldots, (N, j-1), (N, j), (N-1, j), \ldots, (i+1, j), (i, j)$. Our simple algorithm only requires propagation from the top part of the input boundary to the right part of the output boundary, so the call to *Propagate* only passes the appropriate sections of the matrices and distance table M in the appropriate order. Later, in §4, we will use *Propagate* for propagation across the entirity of each boundary and the issue of boundary vertex orders will again be important.

```
Propagate($I[1..n]$, $D[1..n][1..m]$, $O[1..m]$)
    if $m > 0$ then
        {   $p \leftarrow \lfloor(\frac{m+1}{2})\rfloor$
            $O[p] \leftarrow \max_{1 \le x \le n} \{I[x] + D[x][p]\}$
            Determine $\bar{x}$ maximizing the above.
            Propagate($I[1..\bar{x}]$, $D[1..\bar{x}][1..p-1]$, $O[1..p-1]$)
            Propagate($I[\bar{x}..n]$, $D[\bar{x}..n][p+1..m]$, $O[p+1..m]$)

        }

for $z \in [j, i]$ do
    Propagate($C(i, 0..j, z)$, $M[(i, 0..j)][(N..i, j)]$, $C(N..i, j, z)$)
```

A call to *Propagate* correctly sets $O[p] \leftarrow \max_{1 \le x \le n} I[x] + D[x][p]$ for every $p \in [1, m]$ by the same observation used by Apostolico et al. [1]. Namely, if $O[p]$ is maximized for index $\bar{x}$, then the value of $O[q]$ for some $q < p$ must be maximized for an index between 1 and $\bar{x}$, because otherwise the shortest paths through the subgraph of D used by p and q cross and this leads to an easy contradiction of optimality. Similarly, the value of $O[q]$ for some $q > p$ must be maximized for an index between $\bar{x}$ and n. By choosing p as the bisecting index and recursively solving for the points on the left and right, we achieve an efficient divide-and-conquer procedure, as proven later in Theorem 1.

2.4. Given the candidate arrays for vertices on column j, find the candidate arrays for every vertex between column j and i, and record the best-scoring panel twin ending at each.

This last step is easily accomplished using Miller's recurrence. Rigorously stated, we compute $C(x, y, z)$ for all vertices (x, y) between columns j and i and all $z \in [y, \min(i, x)]$. Simultaneously, we determine the cost, $\max_{y \leq z \leq \min(i, x)} \{C(x, y, z)\}$, of the best panel twin ending at each (x, y).

Note that steps 2.2 and 2.3 are not needed for panels $[0, b]$ and $[N-b, N]$ because the region $E(i, j)$ degenerates to a line. During the execution of phase 1 and each execution of step 2.4, the algorithm keeps a record of the best scoring twin so far. We state this as a final phase:

```
3. Output the score of the best twin found over all the stages.
```

Although the problem is to output just the score of the best twin, note that the algorithm actually computes the best-scoring twin to every vertex in the edit graph of A and so can produce more than one twin if desired. Further note that if all that is desired is to examine just adjacent or tandem twins, then it suffices in step 2.4 to take the maximum over just the entries $C(x, y, y)$.

THEOREM 1. *The algorithm above computes twins in $O(N^{2.5} \log^{0.5} N)$ time and $O(N^2)$ space.*

Proof. Phase 1 of the algorithm takes $O(N^3/b)$ time since it involves solving N/b problems each taking time $O(N^2)$. In phase 2, we perform the minor steps N/b times. Steps 2.1 and 2.4 are similar, involving the computation of candidate arrays in rectangles whose sizes are less than $N \times b$. Since the candidate arrays have up to b elements in them, this takes $O(Nb^2)$ for each pair of rectangles or a total of $O(N^2 b)$ time over all panels. Computing a distance table M in step 2.2 takes $O(N^2 \log N)$ time for a total of $O(N^3 \log N/b)$ time over all panels.

It remains to analyze the complexity of step 2.3. If $T(n, m)$ is the time for a call to *Propagate*, then it satisfies the recurrence $T(n, m) \leq T(n_1, (m-1)/2) + T(n_2, (m-1)/2) + O(n)$ for any n_1 and n_2 such that $n_1 + n_2 = n+1$ and boundary condition $T(n, 1) = O(n)$. An easy induction shows that $T(n, m)$ is $O(n \log m + m)$. Thus each invocation of *Propagate* in step 2.3 takes $O(N \log N)$ time for a total time in the step of $O(Nb \log N)$. Over the entire algorithm, the time spent is then $O(N^2 \log N)$.

The overall complexity of the algorithm is determined by choosing b to equalize the $O(N^2 b)$ time spent in steps 2.1 and 2.4 and the $O(N^3 \log N/b)$ time spent in step 2.2. This is achieved when $b = \sqrt{N \log N}$ and gives a running time of $O(N^{2.5} \log^{0.5} N)$. For the space complexity, note that at worst one $DIST$ table needs to be maintained at any given point in the algorithm. □

4. An improved algorithm.

In essence, the algorithm above is designed around computing panel twins where the size of panels is $N^{1/2}$. One hopes that an $O(N^{7/3}$ polylog $N)$ algorithm is possible using panels of size $N^{1/3}$ and, if so, one can get a progression of decreasing times by choosing panels of size $N^{1/K}$, ultimately yielding an $O(N^2 \text{polylog} N)$ algorithm for $K = \log N$. We are indeed able to pull off such a progression, but doing so requires several refinements. First, we have to abandon computing distance tables from scratch for each panel problem. Instead, in a preprocessing step, we produce a mesh of carefully chosen distance tables that permit us to subsequently propagate candidate lists through critical subgraphs in $O(N \log^2 N)$ time. Second, for a basic block size of $N^{1/K}$, we must proceed in K phases, where in the Jth phase the panels are of size $N^{J/K}$. In the Jth phase, we find the twins that are in a given $N^{J/K}$ panel but not in any $N^{(J-1)/K}$ subpanel.

To get a more intuitive feel and motivation for what follows, let us consider the development of an algorithm based on multiples of $N^{1/3}$. Suppose we tried our simple two-level algorithm from the previous section with $N^{2/3}$ panels each of size $N^{1/3}$.

In this case, we would run into two problems. Performing $N^{2/3}$ Smith–Waterman computations at the outer level (step 1) would take $O(N^{8/3})$ total time, and in the inner level we would have to compute $N^{2/3}$ distance tables (step 2.2) for $O(N^{8/3}\log N)$ total time. On the other hand, if we tried $N^{1/3}$ panels each of size $N^{2/3}$, the two previously problematic facets would involve only $O(N^{7/3}\log N)$ total time, which is an improvement over the simple algorithm, but now there would be $N^{2/3}$ candidates to propagate in each lower-level problem, giving rise to a component (steps 2.1 and 2.4) that takes $O(N^{8/3})$ total time. The solution is to go to a three-level scheme: at the outer level, we perform $N^{1/3}$ Smith–Waterman computations to capture all twins that are not in any $N^{2/3}$-panel; at the next level, we iterate over the $N^{1/3}$ panels of size $N^{2/3}$ to capture all twins that are in an $N^{2/3}$-panel but not in any $N^{1/3}$-panel; and at the lowest level, we iterate over the $N^{2/3}$ panels of size $N^{1/3}$ to captures all twins that lie within them. The complexity of the middle level seems at first to still be problematic, as there can be $N^{2/3}$ candidates in an $N^{2/3}$-panel. However, since $N^{1/3}$-twins will be captured by the lowest level, we need not consider every possible start for a twin at this level but simply the best one in each $N^{1/3}$ panel, reducing the number of "pseudocandidates" for this level to $N^{1/3}$. This in effect reduces the total complexity to $O(N^{7/3})$, ignoring for the moment the time for building distance tables. Indeed this scheme generalizes to an arbitrary number of levels. In a K-level scheme, the outer level captures all twins not within any $N^{(K-1)/K}$-panel, the next level captures all twins within an $N^{(K-1)/K}$-panel but not within a nested $N^{(K-2)/K}$-panel, and so on down to a bottom level that captures all twins within an $N^{1/K}$-panel. At level $J < K$, where we capture all twins in an $N^{J/K}$-panel, we need only process the $N^{1/K}$ pseudocandidates that record the best path starting in a nested $N^{(J-1)/K}$-panel, as twins within a panel of that size will be found by the next lower level.

The K-level scheme thus leads to an algorithm requiring $O(KN^{2+1/K})$ time, ignoring the time needed to propagate the $O(N^2)$ pseudocandidates through various regions via distance tables. This is the most difficult obstacle to overcome: we cannot afford to build even the $N^{(K-1)/K}$ distance tables needed for just the lowest level. Instead, we take the $O(N)$ set of all rectangular regions of the edit graph that candidates need to be propagated across, conceptually construct a quad-tree decomposition (see [3] for a description of quad-trees) of these regions, and then build a distance table for each region corresponding to a vertex in the quad-tree decomposition. This permits us to propagate candidates across a region by propagating them through a logarithmic number of precomputed distance tables that partition the region. Moreover, the set of all necessary tables can be efficiently precomputed and stored in $O(N^2 \log N)$ time and space. We thus find a good tradeoff in the time for propagation against the time for constructing distance tables.

With this preamble, we will begin the description of the algorithm. Section 4.1 describes the mesh of distance tables needed to efficiently propagate candidates at every level of the algorithm. Then §4.2 describes the K-level decomposition of the problem and the algorithm based on it. Finally, §4.3 caps the treatment with an analysis of time and space complexity. Throughout, we will assume that $b = N^{1/K}$ is the *basic block size* for some $K \geq 2$ that will be chosen in the final subsection when the competing complexity terms are fully understood. For simplicity, we assume that b is a power of 2 and consequently that N is a power of 2 as well.

4.1. Preprocessing and propagation technique. The preprocessing step consists of computing distance tables for a collection of subproblems. Term an edit-graph vertex $(i,\, j)$ *critical* if it meets the following conditions: (1) i and j are multiples of

b and (2) $0 < j < i < N$. For each critical (i, j), we will compute and associate a number of distance tables. Let $D(p, i, j)$ denote the distance table $DIST_{E(p,i,j)}$ for the square subgraph $E(p, i, j)$ consisting of vertices (x, y) such that $x \in [i, i+p]$ and $y \in [j-p, j]$. In terms of the edit graph, $E(p, i, j)$ is a $p \times p$ square whose upper right corner is the vertex (i, j). We say $E(p, i, j)$ is *cornered at* (i, j). The distance tables to be associated with a given critical vertex correspond to a doubling progression of square subgraphs cornered at the vertex. Specifically, the list of distance tables built for critical vertex (i, j) is $D(b, i, j)$, $D(2b, i, j)$, $D(4b, i, j), \ldots, D(2^x b, i, j)$, where $2^x b$ is the largest power of 2 times the block size that divides both i and j. Let $\max p(i, j) = 2^x b$. Note that because vertices for which $i = j$ are not critical, the largest subgraph for which a distance table is built has $N/4 + 1$ vertices on each side, i.e., $\max p(i, j) \le N/4$. The left half of Figure 3 illustrates the *mesh* of tables just described. Our preprocessing algorithm is simply the following:

```
for $p \leftarrow b, 2b, 4b, \ldots, N/4$ do
    for $i \leftarrow 2p, 3p, 4p, \ldots, N-p$ do
        for $j \leftarrow p, 2p, 3p, \ldots, i-p$ do
        {   $\max p(i, j) \leftarrow p$
            if $p = b$ then
                Compute $D(p, i, j)$ de novo using the algorithm in [1]
            else
                Compute $D(p, i, j)$ by fusing the four tables $D(p/2, i, j)$,
                $D(p/2, i+p/2, j)$, $D(p/2, i, j-p/2)$, and $D(p/2, i+p/2, j-p/2)$
                using the algorithm in [1]
        }
```

In a given iteration of the outer loop, observe that $O((N/p)^2)$ tables are built, and since p doubles with each iteration, $O((N/b)^2)$ tables are built over the entire algorithm. For the iteration where $p = b$, each distance table takes $O(b^2 \log b)$ time to build and occupies $O(b^2)$ space for upper bounds of $O(N^2 \log N)$ time and $O(N^2)$ space for the iteration. Since fusing 4 $p/2 \times p/2$ tables only takes time $O(p^2)$, the time spent in every iteration other than the first is bounded by $O(N^2)$ time and space. There are $O(\log N)$ iterations of the outer loop, for a grand total of $O(N^2 \log N)$ time and space for the preprocessing step.

The *mesh* of $O((N/b)^2)$ distance tables computed in the preprocessing step have been chosen so that rectangular subgraphs of the edit graph relevant to our algorithm can be partitioned into $O(N)$ *mesh squares* for which tables have already been computed. For a critical vertex (i, j), as in §3 let $E(i, j)$, the *critical subgraph cornered at* (i, j), be the rectangular subgraph consisting of all vertices (x, y) such that $x \in [i, N]$ and $y \in [0, j]$. Suppose 2^x is the largest power of 2 dividing i/b and 2^z is the largest power of 2 dividing j/b. Let $mp = \max p(i, j) = 2^{\min(x,z)} b$. The subgraph $E(i, j)$ can be partitioned into mesh squares in the following greedy way:

Case 1: $x < z$. In this case, $E(i, j)$ is partitioned into (a) the row of $mp \times mp$ mesh squares $E(mp, i, mp)$, $E(mp, i, 2mp)$, $E(mp, i, 3mp), \ldots, E(mp, i, j)$ along the top boundary of $E(i, j)$ and (b) the partition of $E(i + mp, j)$ obtained by recursively applying this procedure (unless $i + mp = N$, in which case we are done). Note that $\max p(i + mp, j)$ must be $2mp$ or greater in this case, and so the recursive partitioning of $E(i + mp, j)$ will involve squares of at least twice the size.

Case 2: $x > z$. The situation is symmetric and $E(i, j)$ is partitioned into (a) the recursive partitioning of $E(i, j - mp)$ if $j > mp$ and (b) the column of $mp \times mp$

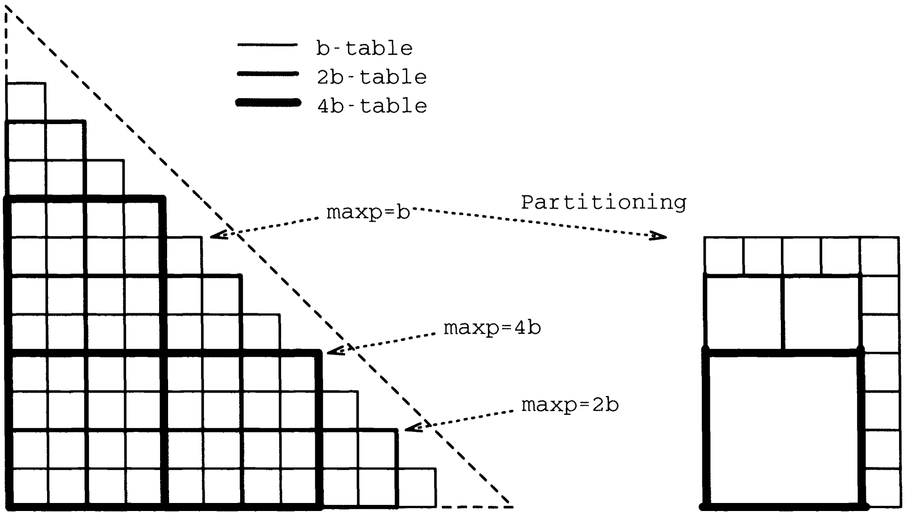


FIG. 3. *Mesh partitioning for preprocessing step.*

mesh squares $E(mp, i, j)$, $E(mp, i{+}mp, j)$, $E(mp, i{+}2mp, j), \ldots, E(mp, N{-}mp, j)$ along the right boundary of $E(i, j)$. Again note that $\max p(i, j - mp)$ must be $2mp$ or greater in this case, and so the recursive partitioning of $E(i, j - mp)$ will involve squares of at least twice the size.

Case 3: $x = z$. Here one "peels" off both a column and row of $mp \times mp$ squares along the top and right boundaries of $E(i, j)$ and then recursively partitions $E(i + mp, j - mp)$ if it is nonempty. Once again, $\max p(i + mp, j - mp)$ must be $2mp$ or greater.

Figure 3 illustrates the partitioning for a particular critical corner. Note that in all cases the partition is into subgraphs cornered at critical vertices and so the distance tables for all the squares constituting the partition of $E(i, j)$ have been computed in the preprocessing step as part of the mesh. Note that the nonrecursive part of the partitioning process employs at most $O(N/mp)$ $mp \times mp$ squares, with a total perimeter of $O(N)$ over all squares. Since mp at least doubles with each level of recursion, it follows that the $E(i, j)$ is partitioned into at most $O(N/mp)$ $mp \times mp$ squares, $O(N/2mp)$ $2mp \times 2mp$ squares, $O(N/4mp)$ $4mp \times 4mp$ squares, ..., and $O(1)$ $N/4 \times N/4$ squares. Therefore, $E(i, j)$ is in general partitioned into at most $O(N/\max p(i, j))$ mesh squares, whose total perimeter sum is at most $O(N \log N)$. This characteristic of the partitioning is important since the running time of a step in the ensuing algorithm is directly proportional to it.

Having now described how to partition the subgraphs $E(i, j)$, we next show how to propagate candidate-list information from the left and top boundaries of $E(i, j)$ to its right and bottom boundaries. For the ensuing algorithm, it will only be necessary, as in the simple algorithm of §3, to propagate candidates from the top boundary to the right boundary, but in the recursive process below we need to solve the more general left and top boundary to right and bottom boundary propagation problem. Recall from §3 that the subprocedure *Propagate*(I[1..n], D[1..n][1..m], O[1..m]) that propagates a vector I through a distance table D to produce the vector O. Propagation through $E(i, j)$ is accomplished by propagating candidates through the mesh partition using *Propagate* to propagate candidates through each mesh square. Recall also that in general the procedure *Propagate* takes input values along the left and top boundaries and produces output values along the bottom and right boundaries of a rectangular subgraph. In the *Mesh_propagate* procedure described below, we will sometimes have separate vectors representing the values at the left boundary and the values at the top boundary. When we want to make a call to *Propagate*, we will combine these vectors using a "concatenate" operator denoted by "·". Similarly, when we want to separately name the values at the bottom and right boundaries, we will also use the concatenate operator and specify two separate output arguments.

```
Mesh_propagate($L[i..N]$, $T[0..j]$, $R[i..N]$, $B[0..j]$)
  { vector $X[mp+1]$, $Y[N+1]$
    $mp \leftarrow$ maxp($i$, $j$)
    if $j \neq mp$ and $mp$ = maxp($i$, $j - mp$) then
      { $R[i..i+mp] \leftarrow L[i..i+mp]$
        for $k \leftarrow mp, 2mp, 3mp, ..., j$ do
          { $X[0..mp] \leftarrow R[i..i+mp]$
            Propagate($X[mp..1] \cdot T[k-mp..k]$, $D(mp, i, k)$, $Y[k-mp..k-1] \cdot$
              $R[i+mp..i]$)
            $Y[k] \leftarrow R[i+mp]$
          }
        if $i < N - mp$ then Mesh_propagate($L[i+mp..N]$, $Y[0..j]$, $R[i+mp..N]$, $B$)
      }

    else
      { if $j > mp$ then Mesh_propagate($L$, $T[0..j-mp]$, $Y[i..N]$, $B[0..j-mp]$)
        $B[j-mp..j] \leftarrow T[j-mp..j]$
        for $k \leftarrow i, i+mp, i+2mp, ..., N-mp$ do
          { $X[0..mp] \leftarrow B[j-mp..j]$
            Propagate($Y[k+mp..k] \cdot X[1..mp]$, $D(mp, k, j)$, $B[j-mp..j-1] \cdot$
              $R[k+mp..k]$)
            $B[j] \leftarrow R[k+mp]$
          }
      }
  }
```

The procedure *Mesh_propagate* propagates the values across $E(i, j)$, where the formal parameters are as follows. Vector $L[i..N]$ contains the input values on the left boundary vertices $(i..N, 0)$ (denoting the sequence $(i, 0), (i+1, 0), \ldots, (N, 0)$). Vector $T[0..j]$ contains the input values on the top boundary vertices $(i, 0..j)$, $R[i..N]$ receives the propagated output values on the right boundary vertices $(i..N, j)$, and $B[0..j]$ receives the output values along the bottom boundary $(N, 0..j)$. Note that the input and output vectors redundantly cover the upper left and lower right corner vertices, i.e., $L[i] \equiv T[0]$ and $R[N] \equiv B[j]$. *Mesh_propagate* propagates the input vectors through the mesh using the recursive decomposition above. For example, if Case 1 is true for (i, j), then the procedure begins by propagating T and $L[i..i+mp]$ through the row of $mp \times mp$ mesh squares to a temporary vector $Y[0..j]$ along the lower output boundary and $R[i..i+mp]$ along the right boundary. Another temporary vector X is used to chain together the propagations through each mesh square via calls to *Propagate*. Note that as in §3, great care is taken to feed to *Propagate* the relevant portions of input and output vectors in the relevant order so as to observe the ordering requirement for distance-table boundary vertices. Once propagation through the row of mesh squares is accomplished, the task is completed by recursively mesh propagating $L[i + mp..N]$ and Y to B and $R[i + mp..N]$. Case 2 is also similarly reflected in the pseudocode of *Mesh_propagate,* and Case 3 is effectively handled by first applying Case 1 and then noting that the recursive call will immediately apply Case 2.

Recall that a call to *Propagate* with an $n \times n$ problem takes $O(n \log n)$ time. Because the mesh partition for $E(i, j)$ contains at most $O(N/mp)$ $mp \times mp$ mesh squares, it then follows that at most $O(N \log mp) = O(N \log N)$ time is spent propagating through mesh squares of this size. Moreover, square size doubles with each recursion, so there are a maximum of $O(\log (N/\max p(i, j))) = O(\log N)$ square sizes in the partition of $E(i, j)$. Thus the total time spent in *Mesh_propagate* is bounded above by $O(N \log^2 N)$.

4.2. The K-phase algorithm. Recall that the basic block size b is $N^{1/K}$, where K is yet to be chosen. For $J \in [1, K]$, let the N/b^J intervals $[0, b^J]$, $[b^J, 2b^J]$, $[2b^J, 3b^J], \ldots, [N - b^J, N]$ constitute the set of *J-panels.* A *J-panel twin* is a panel twin for some J-panel but not for any of the $(J - 1)$-panels within it. Our K-phase algorithm proceeds through phases $J = K, K - 1, \ldots, 1$, where in phase J all J-panel twins are found. Note that the simple algorithm of §3 is a specialization of this approach with $K = 2$: Step 1 found the 2-panel twins of the sole 2-panel $[0, N]$ and step 2 found the 1-panel twins. Note that the general algorithm succeeds in finding every twin, as a twin must be a J-panel twin for some $J \in [1, K]$.

Finding the K-panel twins is an easy generalization of step 1 of the algorithm of §3. Namely, for each partition index $i = b^{K-1}, 2b^{K-1}, \ldots, N - b^{K-1}$, run the Smith–Waterman algorithm on $A[1..i]$ versus $A[i + 1..N]$. For phase $J < K$, the algorithm mimics steps 2.1 through 2.4 of the simple algorithm with two significant modifications. First, step 2.2 is no longer required, as the mesh is computed earlier, and step 2.3 uses *Mesh_propagate* instead of the simpler *Propagate.* The second modification is in the nature of the C-values used in the phase. Since the goal of phase J is to find J-panel twins and no $(J - 1)$-panel twins, one need only keep track of the best path that starts in a given $(J - 1)$-panel. That is, to know if a path is a J-panel twin of a given J-panel, it suffices to only know which of its b $(J - 1)$-subpanels the path starts in. To this end, we introduce the quantity $C^J(i, j, k)$, the best path to (i, j) that begins in the kth $(J-1)$-panel of $[0, N]$, i.e., starts at some vertex between rows kb^{J-1} and $(k + 1)b^{J-1}$. Note that for $J = 1$, the definition of C^J coincides with that of C. The recurrence of §2 is easily modified to describe the computation of C^J below.

For $j \leq i$ and $k \in [\lceil j/b^{J-1} \rceil, \lfloor i/b^{J-1} \rfloor]$,

$$C^J(i, j, k) = \max \begin{cases} C^J(i-1, j-1, k) + \delta(a_i, a_j) & \text{if } i > j > 0 \text{ and } i > kb^{J-1}, \\ C^J(i-1, j, k) + \delta(a_i, \epsilon) & \text{if } i > 0 \text{ and } i > kb^{J-1}, \\ C^J(i, j-1, k) + \delta(\epsilon, a_j) & \text{if } j > 0 \text{ and } i > kb^{J-1}, \\ 0 & \text{if } kb^{J-1} \leq i \leq (k+1)b^{J-1}. \end{cases}$$

With the introduction of C^J, the entire computation of phase $J < K$ can now be described succinctly as follows.

Phase J:

For each J-panel $[j, i] \in [0, b^J], [b^J, 2b^J], [2b^J, 3b^J], \ldots, [N - b^J, N]$:

Step J.1. For $x \in [j, i], y \in [0, j]$, and $z \in [j/b^{J-1}, \lfloor x/b^{J-1} \rfloor]$, compute $C^J(x, y, z)$.

Step J.2. For $z \in [j/b^{J-1}, i/b^{J-1}]$, *Mesh_propagate*($\langle -\infty, -\infty, \ldots, \rangle$, $C^J(i, 0..j, z)$, $C^J(i..N, j, z)$, $\langle -\infty, -\infty, \ldots, \rangle$).

Step J.3. For $y \in [j, i]$, $x \in [y, N]$, and $z \in [\lceil y/b^{J-1} \rceil, \lfloor \min(i, x)/b^{J-1} \rfloor]$, compute $C^J(x, y, z)$ and keep track of the maximum J-panel twin found.

Each quantity $C^J(x, y, z)$ is called a panel-twin candidate. Note that no more than $b+1$ such candidates are computed at any vertex (x, y) in any phase $J < K$. During the execution of the phases, the best twin of any type is recorded and this twin is reported upon completion of all the phases.

4.3. Running-time analysis and space refinement. We can now state and prove the following running-time bound on our algorithm.

THEOREM 2. *The above algorithm computes the optimal pair of twins in time* $O(N^2 \log^2 N)$.

Proof. We will consider the total running time for each activity of the algorithm. First, we have already argued that the preprocessing stage takes time $O(N^2 \log N)$. Phase K of the algorithm invokes the $O(N^2)$ Smith–Waterman algorithm b times for a total of $O(bN^2)$ time. Now consider any other phase $J < K$. Steps J.1 and J.3 compute $O(b)$ panel-twin candidates at each vertex in rectangles of size less than $N \times b^J$ for a total of $O(b^{J+1}N)$ time. Each mesh propagation in Step J.2 takes $O(N \log^2 N)$, as analyzed in §4.1, and is repeated for each of the $O(b)$ panel-twin candidates at the top boundary for a total of $O(bN \log^2 N)$ for the step. The total number of panels processed in phase J is N/b^J, giving a total time for phase J of $O(N^2 (b + \log^2 N/b^{J-1}))$. Summing over all phases gives a grand total for the algorithm of $O(KbN^2 + N^2 \log^2 N)$ time as the $\log^2 N/b^{J-1}$ term telescopes. Choosing $K = \log N$ makes $b = N^{1/K} = 2$ and gives us the bound $O(N^2 \log^2 N)$. Note that the dominant term comes from the cost of propagation through the mesh. □

In a conference version of this paper [4], the authors asked if the space complexity of the algorithm could be reduced from $O(N^2 \mathrm{log} N)$. This was subsequently answered affirmatively by Benson [2], who gave an $O(N^2)$-space algorithm with the same time complexity as ours. While his result uses a distinctive line of attack, we thought that the real essence of the improvement involved abandoning precomputing the entire mesh of distance matrices and instead computing components of the mesh "on the fly" as necessary.

Indeed, we now see that the following simple modification to our algorithm gives us $O(N^2)$ space. Instead of proceeding *recursively* phase by phase, proceed instead *iteratively* in increasing order of 1-panels $[j, i]$, also processing any J-panels, for $J > 1$, that begin at j.

Recall that the basic block size, b is 2. Formally, consider the following outermost structure:

```
for $j \leftarrow 0, b, 2b, \ldots, N - b$ do
  for $p \leftarrow 2^J b = 2^{J+1}$ s.t. $p$ divides $j$ in order of $J$ do
    Perform Steps $J$.1, $J$.2, and $J$.3 on panel $[j, j+p]$,
```

where *no* preprocessing is undertaken. For a given j, we will need the set of distances matrices that partition $E(j+b, j)$. Observe that within this partition, we have the partitions for $E(j+p,j)$ for each of the p that will be processed for a given choice of j. So in order to do mesh propagation in Step $J.2$ of the outline above, it suffices to deliver the mesh partition of $E(j+b,j)$ as we iterate through progressive values of j in the outer loop. Suppose we have the partition for $E(j,j-b)$. To obtain the one for $E(j+b,j)$, discard every distance matrix in the former and not the latter partition and build every distance matrix in the latter but not in the former. We can afford to build a new $p \times p$ matrix "on the fly" using the less efficient $O(p^2 \log p)$ de novo algorithm of Apostolico et al. [1]. At any moment, the space required during this process is easily seen to be $O(N^2)$. The final fact required is to observe that a given distance table $DIST(p,k,h)$ is a part of the partition of $E(j+b,j)$ *exactly* when either (a) $j+b \leq k$ and $h \leq j < h+p$ or (b) $h \leq j$ and $k-p < j+b \leq k$. Thus a given distance table of the mesh will be "demanded" and subsequently discarded at most twice. There are $O((N/p)^2)$ tables of size $p \times p$ all together and $O(\log N)$ choices of p for a total time to build the tables "on the fly" of $O(N^2 \log^2 N)$. Note that whereas before we took $O(N^2 \log N)$ time and space in the preprocessing, we now take $O(N^2 \log^2 N)$ time to build tables in order to use only $O(N^2)$ space.

5. Finding short twins efficiently. In this section, we describe a very simple algorithm to find twins efficiently if we know that the length of the twins is no more than $N^{1-\epsilon}$ for arbitrarily small ϵ. Let $L = N^{1-\epsilon}$ and consider the following recursive approach. Explicitly solve the problem of finding twins between $A[1..N/2]$ and $A[N/2{+}1..N]$. Recursively solve the subproblems of finding twins within $A[1..N/2{+}L]$ and $A[N/2-L..N]$ until the length of the string in the subproblem is no more than $3L$. For subproblems of this size, solve them by using the algorithm of the previous section in time $O(L^{2-\epsilon})$. Letting $T(n)$ be the time to find optimal twins in a string of length n, we get the following recurrence:

$$T(n) = O(n^2) + 2T(L + n/2)$$

with initial condition $T(3L) = O(L^{2-\epsilon})$. Solving this recurrence reveals that $T(N) = O(N^2)$.

This algorithm can be simplified somewhat if it is known that the twins sought are of size no more than $L = N^{2/3-\epsilon}$, in which case the algorithm of §3 is sufficient to handle the base-case problems while still giving a time bound of $O(N^2)$. Finally, if it is known that the twins are not of size more than $O(\sqrt{N})$, then the base cases can simply be handled by the brute-force cubic algorithm (or by Miller's improvement of this algorithm) while still giving an overall running time of $O(N^2)$.

6. Conclusions and open problems. A practical method for detecting repeated regions in DNA and protein sequences needs to find *multiple* (possibly more than two) nonoverlapping regions of maximum alignment score. Since the algorithm presented in the paper finds the best twins ending at each point in the edit graph, it is a relatively simple bookkeeping matter to identify and report more than two regions that have high pairwise alignment scores.

The algorithm presented in this paper is perhaps close to optimal in time complexity, but there is the vexing factor of $\log^2 N$ as opposed to $\log N$. Perhaps some modification of the algorithm would make this improvement. One important concern is also the space complexity of the algorithm. Can this be brought down from $O(N^2 \log N)$ to $O(N^2)$ or even less?

The property exploited by the procedures *Propagate* and *Mesh_Propagate* is that optimal scoring paths do not cross. This property no longer holds when the scoring scheme is generalized further to allow affine or concave gap costs. Thus for these generalized scoring schemes, it is still open whether there is an algorithm that achieves a running time close to the running time of the algorithm in the paper.

REFERENCES

[1] A. Apostolico, M. J. Atallah, L. L. Larmore, and S. McFaddin, *Efficient parallel algorithms for string editing and related problems*, SIAM J. Comput., 19 (1990), pp. 968–988.

[2] G. Benson, *A space efficient algorithm for finding the best nonoverlapping alignment score*, in Proc. 5th Symposium on Combinatorial Pattern Matching, Lecture Notes in Comput. Sci. 807, Springer-Verlag, Berlin, New York, Heidelberg, 1994, pp. 1–14.

[3] R. A. Finkel and J. L. Bentley, *Quad-trees: A data structure for retrieval on composite key*, Acta Inform., 4 (1974), pp. 1–9.

[4] S. Kannan and E. Myers, *An algorithm for locating nonoverlapping regions of maximum alignment score*, in Proc. 4th Symposium on Combinatorial Pattern Matching, Lecture Notes in Comput. Sci. 648, Springer-Verlag, Berlin, New York, Heidelberg, 1993, pp. 74–86.

[5] G. M. Landau and J. P. Schmidt, *An algorithm for approximate tandem repeats*, in Proc. 4th Symposium on Combinatorial Pattern Matching, Lecture Notes in Comput. Sci. 648, Springer-Verlag, Berlin, New York, Heidelberg, 1993, pp. 120–133.

[6] V. I. Levenshtein, *Binary codes of correcting deletions, insertions and reversals*, Soviet Phys. Dokl., 10 (1966), p. 707.

[7] M. G. Main and R. J. Lorentz, *An $O(n \log n)$ algorithm for finding all repetitions in a string*, J. Algorithms, 5 (1984), pp. 422–432.

[8] W. Miller, *An algorithm for locating a repeated region*, private communication.

[9] E. W. Myers, *An $O(ND)$ difference algorithm and its variants*, Algorithmica, 1 (1986), pp. 251–266.

[10] T. F. Smith and M. S. Waterman, *Identification of common molecular sequences*, J. Molecular Biol., 147 (1981), pp. 195–197.

[11] 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. 25, No. 3, pp. 663–696, June 1996

FULL ABSTRACTION AND THE CONTEXT LEMMA*

TREVOR JIM[†] AND ALBERT R. MEYER[‡]

Abstract. It is impossible to add a combinator to PCF to achieve full abstraction for models such as Berry's stable domains in a way analogous to the addition of the "parallel-or" combinator that achieves full abstraction for the familiar complete partial order (cpo) model. In particular, we define a general notion of rewriting system of the kind used for evaluating simply typed λ-terms in Scott's PCF. Any simply typed λ-calculus with such a "PCF-like" rewriting semantics is shown necessarily to satisfy Milner's Context Lemma. A simple argument demonstrates that any denotational semantics that is adequate for PCF, and in which certain simple Boolean functionals exist, cannot be fully abstract for *any* extension of PCF satisfying the Context Lemma. An immediate corollary is that stable domains cannot be fully abstract for any extension of PCF definable by PCF-like rules.

Key words. stable functions, full abstraction, Context Lemma, PCF, standardization

AMS subject classifications. 03B40, 68N15, 68Q40, 68Q50, 68Q55

1. Introduction.

A paradigmatic example of a functional programming language is PCF, Scott's simply typed λ-calculus for recursive functions on the integers [35]. Many categories of denotational meaning are known to *adequately* reflect the computational behavior of PCF in a precise technical sense, namely, a PCF term evaluates to the numeral $\underline{n}$ iff it means the integer n. But typically there are pairs of terms with distinct meanings that nevertheless are computationally indistinguishable in PCF. For example, with the semantics based on complete partial orders (cpos), PCF must be extended with a "parallel-or" combinator in order to express enough computations to be *fully abstract*, i.e., semantical distinctions and computational distinctions between terms coincide [32, 31].

The problem of giving a semantical description of a fully abstract model of unextended PCF remained open for nearly fifteen years (cf. [28, 9, 29, 37]); it has only recently been solved [3, 21]. During that time, efforts to construct spaces of "sequential" functions corresponding to those definable in the original PCF without parallelism led to the discovery of a number of new domains suitable for denotational semantics. Although none are fully abstract for PCF, one motivation for the development of spaces such as the *stable* functions, *bistable* functions, *sequential algorithms* [6, 5, 9, 8, 16], and most recently the *strongly stable* functions [14], was that they captured various aspects of sequentiality and so seemed "closer" to full abstraction for unextended PCF than the popular cpo model.

The stable function model in particular has a simple definition and attractive category-theoretic properties. Its only apparent technical peculiarity is that stable domains of functions are not partially ordered pointwise; in general, the stable ordering strictly refines the pointwise ordering. Nevertheless, just as for the cpo model, the elements of stable domains of type $\sigma \to \tau$ are actually total functions from elements of type σ to elements of type τ. Likewise, there is a natural notion of finite and effective

* Received by the editors December 30, 1991; accpeted for publication (in revised form) November 25, 1994.

[†] Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139 (trevor@theory.lcs.mit.edu). The research of this author was supported by ARO grant DAAL03-89-G-0071.

[‡] Laboratory for Computer Science, Cambridge, Massachusetts Institute of Technology, MA 02139 (meyer@theory.lcs.mit.edu). The research of this author was supported by ONR grant N00014-89-J-1988 and NSF grant 8819761-CCR.

elements of stable domains, and these domains yield an adequate least fixed-point model for PCF. Further, they form a cartesian closed category with solutions for domain equations [6]. This category was also independently discovered and used in constructing a model of polymorphic λ-calculus [17]. So the stable domains seem to offer a setting for a theory for higher-order recursive computation with many of the attractions of the cpo category.

However, one important result about cpos is not known for stable domains, namely, full abstraction with respect to some extension of PCF analogous to the parallel-or extension which Plotkin and Sazonov provided for the cpo model. What might a symbolic-evaluator for an extended PCF look like if it was well matched—fully abstract—with the stable model? We conclude that such an evaluator will have to be unusual looking: it cannot be specified by the kind of term-rewriting-based evaluation rules known for PCF and its extensions.

The significance of this negative result hinges heavily on how drastic we judge it to go beyond the scope of PCF-like rules. It is of course possible that some operational behavior that we declare to be non-PCF-like, in our technical sense, will nevertheless offer a useful extension of PCF for which stable domains are fully abstract. For example, Bloom [11] provides such an extension for complete lattice models, though he goes on to criticize the rather complex algorithmic specification of the combinators in his extension. (The general benefits of structured approaches to operational semantics and connections to full abstraction are discussed in [27, 12].)

To illustrate the generality of our notion of PCF-like rules, we note that the standard extensions of PCF by parallel-or and existential combinators are easily seen to be PCF-like. For example, we can define an evaluator for Plotkin's $\exists$ constant [31] while remaining within a term-rewriting discipline, as follows. Let $p : \iota \rightarrow o$ be an "integer predicate" variable, and use the rules

$$\exists p \rightarrow \mathsf{cond}\,(p\underline{n})\,\mathtt{tt}\,\Omega,$$
$$\exists p \rightarrow \mathsf{cond}\,(p\Omega)\,\Omega\,\mathtt{ff}.$$

The resulting PCF-like language no longer has a confluent rewriting system, though it remains single valued, viz., every term rewrites to at most one numeral. In general, our PCF-like rules need not even be single valued.

A substantial technical contribution of this paper is a simple, modest restriction on the format of rewrite rules which is sufficient to guarantee Milner's Context Lemma [28] for languages defined by such rules. Informally, this "Approximation" Context Lemma requires that if two phrases M and N of the same syntactic functional type yield visibly distinct computational outcomes when used in *some* language context, then there are actual parameters of appropriate argument type such that M and N, each simply *applied* to these arguments, yield visibly distinct computational outcomes. This property, more perspicuously dubbed *operational extensionality* by Bloom [10, 11], has been identified by many authors as technically significant in program semantics [38, 30, 25, 1, 19, 2, 36]. The key to the proof of the Context Lemma is a new Standard Reduction Theorem 4.4 for PCF-like rewrite systems.

Our work borrows much from Bloom [10, 11]. The second author raised the question of whether there is a "reasonable" extension of PCF that would yield a fully abstract evaluator for lattice models [33, 34]. In answering this question, Bloom emphasized how the Context Lemma and full abstraction were incompatible with *single-valued* evaluators for the lattice model. He also characterized a general class of *consistent* rewrite rules that ensured the soundness of the Context Lemma. However, in order to encompass the computational behavior of the $\exists$ combinator, Bloom needed to develop an auxiliary notion of "observation calculi."

Our PCF-like rules are, in an appropriate sense, as powerful as Bloom's observational calculi, and strictly subsume the class of consistent rules. In particular, consistent rules are necessarily confluent and hence single valued; as Bloom remarks [10], introducing a `join` combinator with simple multiple-valued rewrite rules yields a PCF extension both fully abstract for the lattice model and also satisfying the Context Lemma. Our wish to simplify Bloom's criteria while dealing with nonconfluent rewriting systems forced us, instead, to a rather elaborate theory of standard reductions.

As an aside, we also point out that it is questionable whether the (bi)stable and similar domains are closer to full abstraction for PCF. In particular, although some operationally valid equations that fail in the cpo model do hold, for example, in the stable model, we note in Corollary 3.3 that the converse also happens: some equations that hold in the cpo model fail in the stable model. The cpo, stable, and likewise the bistable models thus offer information about the operational behavior of PCF terms that is not apparently comparable, and it is hard to see how to judge which is a more accurate model.

The outline of our argument is as follows. In §2, we formulate the key concepts of observational approximation, adequacy, and full abstraction in a general setting. Then in §3, Theorem 3.2, we give a short proof that any denotational semantics that is adequate for PCF and in which a certain simple Boolean functional exists cannot be fully abstract for extensions of PCF satisfying the Context Lemma. The Boolean functional is obviously not continuous in Scott's sense, but it is stably continuous and so does appear in the stable model. We also formulate a Comparability Context Lemma which applies to the bistable domains. Section 4 gives our general notion of term rewriting systems of the kind used for symbolic evaluation of PCF terms. Then in §5, we show that any such system defines an observational approximation relation that must satisfy the Context Lemma [28]. An immediate corollary is Theorem 5.5, which states that there is no extension of PCF defined by PCF-like rewriting rules for which the stable domain semantics is fully abstract. We state but do not prove a similar result for the bistable domains.

2. Adequacy and full abstraction. Concepts concerning program behavior, such as observational congruence, adequacy, and full abstraction, can usefully be defined in a general setting consisting of the following:

- a set $\mathcal{L}$, called a *language*, whose elements, $M, N, \ldots$, are called *terms*;
- partial operators $C[\cdot]$ on terms called *contexts*; and
- a set $\mathcal{O}$, called a *notion of observation*, whose elements are predicates on terms called *observations*. When an observation is true of a term, the term is said to *yield* the observation.

We will work with languages whose operational behavior is specified by (possibly nondeterministic) symbolic evaluation of terms, so we further assume a binary relation, "evaluates to," on terms. For such languages, $\mathcal{O}_{\text{eval}}$ captures the familiar notion of observing the final output of an evaluation:

$$\mathcal{O}_{\text{eval}} = \{\text{ "evaluates to } O\text{" } \mid O \text{ is an output term}\}.$$

Here the output terms are those terms regarded as observable "output values." These typically include the ground constants (integers, truth values, ...); λ-abstractions and finite lists of output values might also be included.

There are other notions of observation based on evaluation. For instance, $\mathcal{O}_{\text{lazy}}$ consists of the single predicate true of exactly those terms whose evaluation can terminate. And notions of observation can be based on semantics of terms, e.g.,

$$\mathcal{O}_{\text{int}} = \{\text{“has the meaning of } O\text{”} \mid O \text{ is an output term}\}.$$

In this paper, however, we will be mainly concerned with $\mathcal{O}_{\text{eval}}$.

Any notion of observation induces a preordering on terms called *observational approximation.* Intuitively, one term approximates another if, according to the chosen notion of observation, the approximated term exhibits at least as much observable behavior when used in any program as the approximating term.

DEFINITION 2.1. *Let $\mathcal{L}$ be a language with a notion of observation $\mathcal{O}$. A term M* observationally approximates *a term N, written $M \sqsubseteq_{\text{obs}} N$, if for all contexts $C[\cdot]$, whenever $C[M]$ is a term yielding an observation from $\mathcal{O}$, then $C[N]$ is a term yielding it as well. M and N are* observationally congruent, *written $M \equiv_{\text{obs}} N$, iff $M \sqsubseteq_{\text{obs}} N$ and $N \sqsubseteq_{\text{obs}} M$.*

Observational approximation provides precise meaning for questions such as “Does my code meet a specification?” or “Will my new implementation of a module change the behavior of the program?”

In languages like PCF with applicative syntax and a suitable notion of closed terms, analysis of observational approximation can be simplified by appealing to a *Context Lemma.*

DEFINITION 2.2. *Let $\mathcal{L}$ be a language with a notion of observation $\mathcal{O}$. We say a term M* applicatively approximates *a term N, written $M \sqsubseteq_{\text{app}} N$, iff for all vectors of closed terms, $\vec{P}$, whenever $M\vec{P}$ is a term yielding an observation, $N\vec{P}$ is a term yielding it as well. The* Approximation Context Lemma[1] *holds if for all closed terms M and N,*

$$M \sqsubseteq_{\text{app}} N \quad \textit{iff} \quad M \sqsubseteq_{\text{obs}} N.$$

A fundamental result of Milner [28] is that under $\mathcal{O}_{\text{eval}}$ with numerals taken as the output terms, PCF itself, as well as its extension with parallel-or, satisfies the Approximation Context Lemma. We will see later that the Approximation Context Lemma holds for *all* languages defined in a “PCF-like” operational discipline, including, of course, PCF and its familiar extensions.

One method for proving observational approximations is by developing an abstract meaning, $[\![M]\!]$, of a term M that is adequate to determine its observations.

DEFINITION 2.3. *A* meaning function *for a language $\mathcal{L}$ is a function $[\![\cdot]\!]$ from terms M to values $[\![M]\!]$ in some set, partially ordered by a relation $\sqsubseteq$. A meaning function is* compositional *iff for all terms M and N and contexts $C[\cdot]$, if $[\![M]\!] \sqsubseteq [\![N]\!]$ and $C[M]$ is a term, then $C[N]$ is a term and $[\![C[M]]\!] \sqsubseteq [\![C[N]]\!]$.*

A meaning function is adequate[2] *for a notion of observation $\mathcal{O}$ iff for all terms M and N and all observations obs $\in \mathcal{O}$,*

$$\big([\![M]\!] \sqsubseteq [\![N]\!] \textit{ and } obs(M)\big) \quad \textit{implies} \quad obs(N).$$

[1] In particular when $\mathcal{O}$ is $\mathcal{O}_{\text{eval}}$, Bloom [10] calls this “operational extensionality” while Milner [28] uses simply “the Context Lemma.” We use the more descriptive “Approximation Context Lemma” because we will later consider Context Lemmas that are not based on approximation.

[2] As with the Context Lemma, we might more descriptively call this “approximation adequate”; but we will use only the version of adequacy based on approximation and call it simply adequacy for brevity.

Adequacy and compositionality guarantee that the meanings accurately predict observational approximation.

LEMMA 2.4. *A compositional meaning function* $[\![\cdot]\!]$ *is adequate for a notion of observation iff for all terms* M *and* N,

$$[\![M]\!] \sqsubseteq [\![N]\!] \quad \textit{implies} \quad M \sqsubseteq_{\text{obs}} N.$$

The ordering on adequate meanings may be strictly finer than observational approximation. In the ideal situation, known as *full abstraction*, the two orderings coincide.

DEFINITION 2.5. *Let* $[\![\cdot]\!]$ *be a meaning function for a language* $\mathcal{L}$ *with a notion of observation* $\mathcal{O}$. *We say* $[\![\cdot]\!]$ *is* approximation fully abstract[3] *if for all terms* M *and* N,

$$[\![M]\!] \sqsubseteq [\![N]\!] \quad \textit{iff} \quad M \sqsubseteq_{\text{obs}} N.$$

It is equationally fully abstract *if for all* M *and* N,

$$[\![M]\!] = [\![N]\!] \quad \textit{iff} \quad M \equiv_{\text{obs}} N.$$

Approximation full abstraction trivially implies adequacy for compositional meaning functions. Assuming that each output term evaluates to itself, it follows immediately that if $[\![\cdot]\!]$ is adequate for $\mathcal{O}_{\text{eval}}$ and $[\![O]\!] \sqsubseteq [\![M]\!]$, then M evaluates to O, for any output term O. If, in addition, the meaning function is *sound* for the evaluator, we easily obtain a familiar (cf. [27]) alternate characterization of adequacy.

DEFINITION 2.6. *A meaning function* $[\![\cdot]\!]$ *is* sound *for an "evaluates to" relation if for all terms* M *and* N,

$$M \textit{ evaluates to } N \quad \textit{implies} \quad [\![M]\!] = [\![N]\!].$$

LEMMA 2.7. *A sound, compositional meaning function* $[\![\cdot]\!]$ *is adequate for* $\mathcal{O}_{\text{eval}}$ *iff*

$$[\![O]\!] = [\![M]\!] \quad \textit{iff} \quad M \textit{ evaluates to } O,$$

for all terms M *and output terms* O.

This paper focuses specifically on the language PCF and its extensions. The precise (usual) definitions of PCF syntax and semantics appear in Appendix A, and we provide only a quick review here.

PCF is a simply typed λ-calculus with Boolean and natural number ground types, numerals $\underline{n}$ for $n \geq 0$, Boolean constants `tt` and `ff`, and simple arithmetic, recursion, and conditional operators. The evaluation relation $\twoheadrightarrow$ of the language is given by term-rewriting rules.

DEFINITION 2.8. *An* extension of PCF *is a simply typed language together with a set of rewrite rules, such that the types, typed constants, and rewrite rules of the extension include those of PCF. The extension is* conservative *iff for all PCF terms* M, *and all terms* N *in the extension,*

$$M \twoheadrightarrow_{\text{extended}} N \quad \textit{iff} \quad M \twoheadrightarrow_{\text{PCF}} N.$$

Observational congruence, adequacy, etc. for PCF and its extensions will be defined with respect to $\mathcal{O}_{\text{eval}}$, where we take the rewriting relation $\twoheadrightarrow$ as the "evaluates to" relation, and the output terms are the ground constants `tt`, `ff`, and $\underline{n}$ for $n \geq 0$.

The results of the next section, which examines full abstraction for models of extensions of PCF, require that we prove facts about the meanings of terms while

[3] Stoughton [37] calls this "inequationally fully abstract."

knowing very little about the extensions or the models. We will only have adequacy, conservativity, and a few other assumptions to work with. The following lemma shows that this gives us enough to reason about the unextended terms of the language.

LEMMA 2.9. *If a model is adequate for a conservative extension of PCF, then it is also adequate for PCF.*

Proof. Suppose a model $[\![\cdot]\!]$ is adequate for a conservative extension of PCF, and $[\![M]\!] \sqsubseteq [\![N]\!]$ for some PCF terms M and N. All models are compositional, so $[\![C[M]]\!] \sqsubseteq [\![C[N]]\!]$ for any PCF context $C[\cdot]$. So for any ground PCF constant c, if $C[M] \twoheadrightarrow_{\text{extended}} c$, then $C[N] \twoheadrightarrow_{\text{extended}} c$ by adequacy. And then by conservativity, if $C[M] \twoheadrightarrow_{\text{PCF}} c$, then $C[N] \twoheadrightarrow_{\text{PCF}} c$. Hence, $M \sqsubseteq_{\text{obs}}^{\text{PCF}} N$. □

We will further require that our models be sound, and that the ground types o and ι be interpreted as the flat cpos $\{tt, f\!f\}_\perp$ and $\{0, 1, \ldots\}_\perp$, with the standard interpretation of `tt`, `ff`, and the numerals $\underline{n}$. Such models will be called *models with Booleans* (though they are indeed also models with integers). Two models with Booleans of particular interest are the cpo model $\mathcal{C}[\![\cdot]\!]$ and the stable model $\mathcal{S}[\![\cdot]\!]$. Both models are adequate but not fully abstract for PCF.

The additional information about the ground types of models with Booleans is in fact enough to determine the meanings of ground PCF terms.

LEMMA 2.10. *The meaning of any closed PCF term of ground type is the same in all models with Booleans that are adequate for PCF.*

Proof. Let M be a closed PCF term of type o (the case $M : \iota$ is similar). In PCF, exactly one of the following holds: (1) $M \twoheadrightarrow_{\text{PCF}} \texttt{tt}$; (2) $M \twoheadrightarrow_{\text{PCF}} \texttt{ff}$; or (3) neither (1) nor (2) holds. And by Lemma 2.7, $M \twoheadrightarrow_{\text{PCF}} \texttt{tt}$ iff $[\![M]\!] = [\![\texttt{tt}]\!] = tt$ for *any* model with Booleans $[\![\cdot]\!]$ adequate for PCF. Similarly, cases (2) and (3) imply $[\![M]\!] = f\!f$ and $[\![M]\!] = \perp$ respectively. □

Thus we can use any particular adequate model with Booleans, like the familiar cpo model, to discover the meaning of ground PCF terms for arbitrary adequate models with Booleans. We have less to say about terms of higher type. But the following notions are useful.

DEFINITION 2.11. *Let τ be a first-order type, that is, a type of the form $\sigma_1 \to \cdots \to \sigma_n$, where σ_j is a ground type for $1 \leq j \leq n$. Let $[\![\cdot]\!]_i$ for $i = 1, 2$ be type frames such that $\sqsubseteq_1$ on $[\![\sigma_j]\!]_1$ equals $\sqsubseteq_2$ on $[\![\sigma_j]\!]_2$, and let $f_i \in [\![\tau]\!]_i$. Then f_1* pointwise approximates *f_2, written $f_1 \sqsubseteq_{\text{pnt}} f_2$, iff for all $d_j \in [\![\sigma_j]\!]_1$,*

$$f_1(d_1) \cdots (d_n) \sqsubseteq_1 f_2(d_1) \cdots (d_n).$$

It follows immediately from Lemma 2.10 that the functions that are the meanings of a PCF term of first-order type agree *pointwise* in all models with Booleans that are adequate for PCF. So we can use the meaning of a first-order PCF term in some particular model to reason about its meaning in any adequate model with Booleans.

However, pointwise equality is not quite the same as equality of functions. For example, consider the conditional constant $\textsf{cond}_o : o \to o \to o \to o$. Now $\mathcal{S}[\![\textsf{cond}_o]\!] \equiv_{\text{pnt}} \mathcal{C}[\![\textsf{cond}_o]\!]$. But the stable domain does not contain parallel-or, so the stable and cpo meanings of $o \to o \to o$ are different. Thus, $\mathcal{S}[\![\textsf{cond}_o]\!] \neq \mathcal{C}[\![\textsf{cond}_o]\!]$ since the two functions have different codomains.

Nevertheless, it follows immediately from the definitions that pointwise approximation has the following useful property.

LEMMA 2.12. *Let $[\![\cdot]\!]$ be a model with Booleans that is adequate for PCF, and let M and N be closed PCF terms of first-order type. Then*

$$[\![M]\!] \sqsubseteq_{\text{pnt}} [\![N]\!] \quad \textit{implies} \quad M \sqsubseteq_{\text{app}} N.$$

3. Failures of full abstraction. Our first theorem hinges on the presence of certain simple functionals over the Booleans.

DEFINITION 3.1. *Let True be the constant tt function on the flat Booleans, and True! be the strict constant tt function. A* true-separator *is a function f satisfying*

$$f(\mathit{True}) = \mathit{tt},$$
$$f(\mathit{True!}) = \mathit{ff}.$$

THEOREM 3.2. *Let $[\![\cdot]\!]$ be a model with Booleans that is adequate for some conservative extension of PCF satisfying the Approximation Context Lemma. If $[\![\cdot]\!]$ contains a true-separator, it is not equationally fully abstract.*

Proof. Define the terms

$$\mathtt{True} \overset{\text{def}}{\equiv} \lambda x.\mathtt{tt},$$
$$\mathtt{True!} \overset{\text{def}}{\equiv} \lambda x.\mathtt{cond}\, x\, \mathtt{tt}\, \mathtt{tt}.$$

By the definition of a model with Booleans, $[\![\mathtt{True}]\!] = \mathit{True}$. And by Lemma 2.10, $[\![\mathtt{cond}]\!] \equiv_{\text{pnt}} \mathcal{C}[\![\mathtt{cond}]\!]$, so by definition of model with Booleans, we have $[\![\mathtt{True!}]\!] = \mathit{True!}$. Then $\mathtt{True!} \sqsubseteq_{\text{app}} \mathtt{True}$ by Lemmas 2.9 and 2.12. So by the Approximation Context Lemma, $\mathtt{True!} \sqsubseteq_{\text{obs}} \mathtt{True}$.

We conclude that there is no term P defining a true-separator; otherwise $\mathtt{True!}$ and $\mathtt{True}$ yield distinct observations in the context $(P\ [\cdot])$, contradicting the fact that $\mathtt{True!} \sqsubseteq_{\text{obs}} \mathtt{True}$.

However, we can define a true-separator *detector*, D, as follows:

$$D \overset{\text{def}}{\equiv} \lambda x.\mathtt{cond}\,(x\,\mathtt{True})\,(\mathtt{cond}\,(x\,\mathtt{True!})\,\Omega^o\,\mathtt{tt})\,\Omega^o,$$

where Ω^o is the divergent term $(\mathtt{Y}_o(\lambda z^o.z))$. By Lemma 2.10, $[\![\Omega^o]\!] = \mathcal{C}[\![\Omega^o]\!] = \bot$, and so

$$[\![D]\!](f) = \begin{cases} \mathit{tt} & \text{if } f \text{ is a true-separator,} \\ \bot & \text{otherwise.} \end{cases}$$

Now $[\![\lambda x.\Omega^o]\!]$ is the constant $\bot$ function, so $[\![D]\!] \neq [\![\lambda x.\Omega^o]\!]$, since they differ exactly on arguments that are true-separators. But since true-separators are not definable by terms, D and $\lambda x.\Omega^o$ are applicatively congruent. Then by the Approximation Context Lemma, they are observationally congruent, contradicting equational full abstraction. □

COROLLARY 3.3. *If a stable function model with Booleans is adequate for a conservative extension of PCF that satisfies the Approximation Context Lemma, then the model is not equationally fully abstract.*

Proof. Every stable function model with Booleans contains a true-separator *truesep*, defined as follows:

$$\mathit{truesep}(g) = \begin{cases} \mathit{tt} & \text{if } g = \mathit{True}, \\ \mathit{ff} & \text{if } g = \mathit{True!}, \\ \bot & \text{otherwise.} \end{cases} \qquad \square$$

COROLLARY 3.4. *The PCF equations valid in the stable model do not include those valid in the cpo model.*

Proof. Just note that $\mathcal{C}[\![D]\!] = \mathcal{C}[\![\lambda x.\Omega^o]\!]$, but $\mathcal{S}[\![D]\!] \neq \mathcal{S}[\![\lambda x.\Omega^o]\!]$. □

Our proof of Corollary 3.3 of course takes advantage of the notable fact that the stable ordering of functions differs from the pointwise ordering, e.g., the pair of functions *True* and *True!* are ordered pointwise but are stable-incomparable. In fact, the first few lines of the proof of Theorem 3.2 already show that *in*equational full abstraction is incompatible with the Approximation Context Lemma for any model in which *True* and *True!* are incomparable; the rest of the proof justifies the stronger conclusion that *equational* full abstraction fails as well.

We remark that the authors of [14] have informed us that their *strongly stable* models are adequate models with Booleans for PCF and that *truesep* is strongly stable, so Theorem 3.3 and Corollary 3.4 hold for strongly stable models.

Berry realized that altering the pointwise ordering of functions caused difficulties, and he proposed from the start an additional *bistable* model which combines stability with the pointwise ordering. Since the counterexample of Corollary 3.3 relies on the nonpointwise stable ordering, it does not apply to the bistable model.

There is, however, an interesting counterexample to the full abstraction of the bistable model that provides a starting point for extending our results. The counterexample, noted in [16], has its roots in the fundamental motivation behind stable models, viz., to eliminate elements like parallel-or. Consider the following definition.

DEFINITION 3.5. *Let lor be the or-function that is strict in its left argument, and ror be the or-function that is strict in its right argument. An* or-separator *is a function f satisfying*

$$f(lor) = tt,$$
$$f(ror) = ff.$$

The cpo model contains a parallel-or function which bounds the left- and right-strict or-functions, and thus, by monotonicity, cannot contain an or-separator. Since the cpo model is adequate for PCF, an or-separator is not definable in PCF. On the other hand, the stable and bistable models do not contain parallel-or, and in fact, both contain or-separators.

Thus in extending the results to the bistable model, one might try to use an or-separator in the role played by the true-separator in the stable case. Since neither *lor* nor *ror* applicatively approximates the other, an argument based on the Approximation Context Lemma will not work; but a similar argument based on a notion of *observational comparability* does apply.

DEFINITION 3.6. *Let $\mathcal{L}$ be a language with a notion of observation $\mathcal{O}$. Terms M and N are* directly comparable *provided the set of observations yielded by M is setwise comparable to that yielded by N. The terms are* observationally comparable*, written $M \sim_{\text{obs}} N$, if for all contexts $C[\cdot]$, the terms $C[M]$ and $C[N]$ are directly comparable. They are* applicatively comparable*, written $M \sim_{\text{app}} N$, if for all vectors $\vec{P}$ of closed terms, $M\vec{P}$ and $N\vec{P}$ are directly comparable. $\mathcal{L}$ with $\mathcal{O}$ is said to* satisfy the Comparability Context Lemma *if for all closed terms M and N,*

$$M \sim_{\text{app}} N \quad \textit{iff} \quad M \sim_{\text{obs}} N.$$

THEOREM 3.7. *Let $[\![\cdot]\!]$ be a model with Booleans that is adequate for some conservative extension of PCF satisfying the Comparability and Approximation Context Lemmas. If $[\![\cdot]\!]$ contains an or-separator, it is not equationally fully abstract.*

Proof. Consider the terms

$$\mathtt{lor} \overset{\text{def}}{\equiv} \lambda xy.\mathtt{cond}\, x\, \mathtt{tt}\, (\mathtt{cond}\, y\, \mathtt{tt}\, \mathtt{ff}),$$

$$\mathtt{ror} \overset{\mathrm{def}}{\equiv} \lambda xy.\mathtt{cond}\, y\, \mathtt{tt}\, (\mathtt{cond}\, x\, \mathtt{tt}\, \mathtt{ff}).$$

By Lemmas 2.9, 2.10, and 2.12, we have $[\![\mathtt{lor}]\!] = \mathit{lor}$, $[\![\mathtt{ror}]\!] = \mathit{ror}$, and $\mathtt{lor} \sim_{\mathrm{app}} \mathtt{ror}$. So by the Comparability Context Lemma, $\mathtt{lor} \sim_{\mathrm{obs}} \mathtt{ror}$.

We conclude that there is no term P defining an or-separator; otherwise, `lor` and `ror` yield distinct observations in the context $(P\ [\cdot])$, contradicting the fact that $\mathtt{lor} \sim_{\mathrm{obs}} \mathtt{ror}$.

However, we can define an or-separator *detector* as follows:

$$D \overset{\mathrm{def}}{\equiv} \lambda x.\mathtt{cond}\, (x\ \mathtt{lor})\, (\mathtt{cond}\, (x\ \mathtt{ror})\, \Omega^o\, \mathtt{tt})\, \Omega^o.$$

By Lemma 2.10,

$$[\![D]\!](f) = \begin{cases} \mathtt{tt} & \text{if } f \text{ is an or-separator,} \\ \bot & \text{otherwise.} \end{cases}$$

Now $[\![D]\!] \neq [\![\lambda x.\Omega^o]\!]$, since they differ exactly on arguments that are or-separators. But since or-separators are not definable by terms, D and $[\![\lambda x.\Omega^o]\!]$ are applicatively congruent. Then by the Approximation Context Lemma, they are observationally congruent, contradicting equational full abstraction. □

COROLLARY 3.8. *If a bistable model with Booleans is adequate for a conservative extension of PCF that satisfies the Comparability and Approximation Context Lemmas, then the model is not equationally fully abstract.*

Proof. Every bistable model with Booleans contains an or-separator *orsep*, defined as follows:

$$\mathit{orsep}(g) = \begin{cases} \mathit{tt} & \text{if } g = \mathit{lor}, \\ \mathit{ff} & \text{if } g = \mathit{ror}, \\ \bot & \text{otherwise.} \end{cases} \quad □$$

COROLLARY 3.9 (see [23]). *The PCF equations valid in the bistable model do not include those valid in the cpo model.*

Proof. Just note that $\mathcal{C}[\![D]\!] = \mathcal{C}[\![\lambda x.\Omega^o]\!]$, but $\mathcal{B}[\![D]\!] \neq \mathcal{B}[\![\lambda x.\Omega^o]\!]$, where $\mathcal{B}[\![\cdot]\!]$ is the bistable model of [6]. □

The PCF-like languages, defined in the next section, do not satisfy the Comparability Context Lemma. In fact, an or-separator constant can defined through the following PCF-like rules:

$$\mathtt{orsep}(f) \to \mathtt{cond}\, (f\ \mathtt{tt}\ \Omega^o)\, (\mathtt{cond}\, (f\ \mathtt{ff}\ \mathtt{tt})\, (\mathtt{cond}\, (f\ \mathtt{ff}\ \mathtt{ff})\, \mathtt{tt}\, \Omega^o)\, \Omega^o)\, \Omega^o,$$
$$\mathtt{orsep}(f) \to \mathtt{cond}\, (f\ \Omega^o\ \mathtt{tt})\, (\mathtt{cond}\, (f\ \mathtt{tt}\ \mathtt{ff})\, (\mathtt{cond}\, (f\ \mathtt{ff}\ \mathtt{ff})\, \mathtt{ff}\, \Omega^o)\, \Omega^o)\, \Omega^o.$$

Thus we will have to restrict the class of rules we consider if we wish to apply Theorem 3.7. The *consistent* rules of Bloom [11] are an important, natural candidate for the restricted class. We do not know whether the Comparability Context Lemma holds for them. However, we can prove that an or-separator is not definable in consistent systems by a method involving a notion of comparability based on logical relations, as we indicate at the end of the next section.

4. PCF-like rewrite systems. Symbolic evaluators for PCF terms are often presented as term-rewriting systems. In this section, we give the basic definitions for such systems, and give our criteria for calling such a system "PCF-like." Our evaluator for PCF is given in Appendix A.

A *rewrite rule* is a pair $l \to r$ of terms of the same type, such that the free variables of the right-hand side r are included in those of the left-hand side l. We write $M \overset{\Delta}{\to}_\pi N$ if for some subterm Δ of M, $\Delta \to \Delta'$ is an instance of the rule π, and N is obtained from M by replacing Δ with Δ'. We will omit Δ or π as convenient.

Since all of our languages are simply typed λ-calculi, we will always include β-reduction in the rewrite rules of the language. Additionally, we may specify some set Θ of *δ-rules* defining the behavior of the constants. Together, Θ and β define the *rewriting relation* $\to_{\Theta,\beta}$ on the language $\mathcal{L}$. We omit Θ and β when they can be recovered from context.

The δ-rules of PCF have a particularly simple form.

DEFINITION 4.1. *A* linear ground δ-rule *is a rewrite rule of the form*

$$\delta m_1 m_2 \cdots m_n \to P,$$

where each m_i is either a ground constant c_i or a variable x_i. The variables x_i must be distinct. A PCF-like rewrite system *is a language $\mathcal{L}$ together with a set Θ of linear ground δ-rules on the constants of $\mathcal{L}$.*

Note that this definition of "PCF-like" is meant to be generous. In particular, although the system for pure, unextended PCF is both single valued—every term reduces to at most one constant—and confluent, PCF-like systems in general may be multiple valued and nonconfluent.

An interesting example of a multiple-valued PCF-like system arises in [10]. There, Bloom defines an extension of PCF that is both fully abstract and denotationally universal for the lattice model of PCF. The key to the construction amounts to the addition of operators $\top : o$ and $\texttt{join} : o \to o \to o$ with rules

$$\begin{array}{rcll} \texttt{join}\, x\, y & \to & x, & \\ \texttt{join}\, x\, y & \to & y, & \\ \texttt{join}\, \underline{n_1}\, \underline{n_2} & \to & \top, & \underline{n_1} \neq \underline{n_2}, \\ \top & \to & \underline{n}, & n \geq 0. \end{array}$$

Plotkin [31] extended parallel PCF by an existential operator, $\exists : (\iota \to o) \to o$, to achieve a language that is fully abstract and denotationally universal for the cpo model. There, $\exists$ is defined by the deductive rules

$$\frac{p\underline{n} \twoheadrightarrow \texttt{tt}}{\exists p \to \texttt{tt}}, \qquad \frac{p\Omega \twoheadrightarrow \texttt{ff}}{\exists p \to \texttt{ff}},$$

where $\twoheadrightarrow$ is the reflexive transitive closure of $\to$. Because he wanted to be able to specify constants like $\exists$, Bloom [11] introduced *observation calculi* as a definition of "PCF-like" deductive rules.

But note that if we give up confluence, it is possible to define an $\exists$ constant while remaining in a term-rewriting discipline.[4] One such definition was given in the introduction; we provide here a second implementation, which uses the parallel-or combinator $\texttt{por}$.

$$\exists p \to \texttt{por}\,(p0)\,\big(\exists(\lambda x.p(\texttt{succ}\,x))\big),$$
$$\exists p \to \texttt{cond}\,(p\Omega)\,\texttt{tt}\,\texttt{ff}.$$

This kind of rewriting is more straightforward, but actually as powerful as the deductive discipline.

[4] The question of whether $\exists$ can be defined in a *confluent* term-rewriting discipline remains open.

Since PCF-like systems are not confluent in general, we will not be able to use confluence in our proof of the Context Lemma. Instead we will rely on a *Standardization Theorem*, which states that if a term M rewrites to a term N, then there is a "standard" reduction from M to N. Thus we only need consider these standard reductions in our proof.

Typically, the standard reductions are a class of reductions with a particularly nice structure. For instance, in the pure, typed λ-calculus, a standard reduction is one in which redexes are contracted from left to right.

The definition of standard reductions in PCF-like rewrite systems is more complicated because they admit the upwards creation of redexes, cf. [20]. However, there is a simple inductive characterization of those standard reductions that end at a ground constant. This will be sufficient to follow the proof of the Context Lemma given in the next section, so we defer the general definition of standard reductions and the proof of the Standardization Theorem to Appendix C.

Before defining the standard reductions to ground constants, we introduce some useful notation. Consider the set of indices

$$\{\, i \mid m_i \text{ is a constant } c_i \text{ in rule } \theta : \delta\vec{m} \to P \,\}.$$

These indices identify what we call the *critical* arguments of θ, since the rule θ applies to a term $\delta\vec{Q}$ iff $Q_i \equiv c_i$ for i in the set. For expository purposes it will be convenient to separate the critical and noncritical arguments of a constant δ (relative to some linear ground δ-rule θ).

NOTATION 4.2. *Let $\theta : \delta\vec{m} \to P$ be a linear ground δ-rule with j critical arguments and k noncritical arguments. Then for vectors $\vec{A} \equiv A_1 \cdots A_j$ and $\vec{B} \equiv B_1 \cdots B_k$, we let*

$$\delta_\theta\langle \vec{A}, \vec{B}\rangle \overset{\text{def}}{\equiv} \delta\vec{Q},$$

where $\vec{Q}$ is the interleaving of $\vec{A}$ and $\vec{B}$ such that the A_i's appear at the critical indices of $\vec{Q}$. We drop the subscript θ when it can be recovered from context.

Note that we do not require that $\delta\vec{Q}$ be an instance of $\delta\vec{m}$; we will want to use the $\delta\langle\cdot,\cdot\rangle$ notation on terms that we anticipate becoming θ-redexes over the course of a reduction.

In this notation, we write linear ground δ-rules as

$$\theta : \delta\langle\vec{c},\vec{x}\rangle \to P$$

or even

$$\theta : \delta\langle\vec{c},\vec{x}\rangle \to P(\vec{x})$$

when we wish to make the dependence of P on $\vec{x}$ explicit.

DEFINITION 4.3. *The* standard reductions to ground constants *in a PCF-like rewrite system are defined inductively as follows. We will write $M \twoheadrightarrow_s c$ for a standard reduction of a term M to a ground constant c.*

- *If c is a ground constant, then the 0-step reduction $c \twoheadrightarrow c$ is standard.*
- *If $M_1, M_2, \ldots, M_n$ are terms, and c is a ground constant, then a reduction*

$$\begin{aligned}(\lambda x M_1)M_2M_3\cdots M_n &\to_\beta M_1[x := M_2]M_3\cdots M_n\\ &\twoheadrightarrow_s c\end{aligned}$$

is standard.

- *If $C_1, C_2, \ldots, C_n, \vec{D}, \vec{E}$ are terms and $c, c_1, c_2, \ldots, c_n$ are ground constants, then a reduction of the following form is standard:*

$$\begin{array}{lllll}
\boldsymbol{\sigma}_1 : & \delta_\theta\langle C_1 C_2 \cdots C_n, \vec{D}\rangle \vec{E} & \twoheadrightarrow & \delta_\theta\langle c_1 C_2 \cdots C_n, \vec{D}\rangle \vec{E} \\
\boldsymbol{\sigma}_2 : & & \twoheadrightarrow & \delta_\theta\langle c_1 c_2 \cdots C_n, \vec{D}\rangle \vec{E} \\
\vdots & & \twoheadrightarrow & \cdots \\
\boldsymbol{\sigma}_n : & & \twoheadrightarrow & \delta_\theta\langle c_1 c_2 \cdots c_n, \vec{D}\rangle \vec{E} \\
 & & \rightarrow_\theta & P_\theta(\vec{D})\vec{E} \\
 & & \twoheadrightarrow_s & c,
\end{array}$$

where for $1 \le i \le n$, the subreduction $\boldsymbol{\sigma}_i$ consists of a standard reduction from the subterm C_i to the ground constant c_i.

THEOREM 4.4 (Standardization). *For any PCF-like rewrite system, if $M \twoheadrightarrow N$, then there is a standard reduction $M \twoheadrightarrow_s N$.*

Note that if we require our rules to be *nonoverlapping*, then they are a special case of *orthogonal* rewrite systems, for which both confluence and standardization have been known for some time [20]. Similarly, confluence and standardization have been known for the systems of Bloom [11], which restrict our systems by allowing only so-called *consistent* overlaps at the root. However, it is not clear whether $\exists$ can be defined in such systems, and we certainly lose the ability to define interesting nonconfluent systems, such as PCF extended with `join`.

5. The Context Lemma. Once standardization is known, the Context Lemma can be proved by a straightforward adaptation of Bloom's proof for his observation calculi [11]. First, we recall the following basic facts about substitutions.

LEMMA 5.1 (Substitution Lemma). *If $x \not\equiv y$ and $y \notin \mathrm{FV}(L)$, then*

$$M[x := L][y := N[x := L]] \equiv M[y := N][x := L].$$

LEMMA 5.2. *If $x \notin \mathrm{FV}(P)$, then*

$$P[\vec{y} := \vec{N}[x := M]] \equiv (P[\vec{y} := \vec{N}])[x := M].$$

The Context Lemma will follow immediately from this next result.

LEMMA 5.3. *Suppose C is a ground term, c is a ground constant, M and N are closed terms of the same type, and $M \sqsubseteq_{\mathrm{app}} N$. If $C[x := M] \twoheadrightarrow c$, then $C[x := N] \twoheadrightarrow c$.*

Proof. By Standardization, $C[x := M] \twoheadrightarrow_s c$. We show $C[x := N] \twoheadrightarrow c$ by induction on the length of the reduction $C[x := M] \twoheadrightarrow_s c$.

1. The only reduction $C[x := M] \twoheadrightarrow_s c$ of length zero is $c \twoheadrightarrow c$. Then one of the following holds:
   (a) $C \equiv c$. Then clearly $C[x := N] \equiv c \twoheadrightarrow c$.
   (b) $C \equiv x$ and $M \equiv c$. Here $C[x := N] \twoheadrightarrow c$ because $M \sqsubseteq_{\mathrm{app}} N$.

For the induction, we consider subcases on the form of C.

2. $C \equiv (\lambda y C_1) C_2 \cdots C_n$. Assume $x \not\equiv y$ (the case $x \equiv y$ is similar). Since M is closed, we have

$$C[x := M] \equiv \big(\lambda y (C_1[x := M])\big) C_2[x := M] \cdots C_n[x := M].$$

Then the reduction $C[x := M] \twoheadrightarrow_s c$ is of the form

$$\begin{aligned}
C[x := M] &\equiv \big(\lambda y (C_1[x := M])\big) C_2[x := M] \cdots C_n[x := M] \\
&\rightarrow_\beta (C_1[x := M])\big[y := C_2[x := M]\big] C_3[x := M] \cdots C_n[x := M] \\
&\twoheadrightarrow_s c.
\end{aligned}$$

By the Substitution Lemma,

$$(C_1[x := M])\,[y := C_2[x := M]] \equiv (C_1[y := C_2])[x := M],$$

so our reduction can be rewritten

$$\begin{aligned} C[x := M] &\equiv \big((\lambda y C_1)C_2 \cdots C_n\big)[x := M] \\ &\to_\beta \big((C_1[y := C_2])C_3 \cdots C_n\big)[x := M] \\ &\twoheadrightarrow_s c. \end{aligned}$$

Now by β-reduction, the fact that N is closed, and the Substitution Lemma,

$$\begin{aligned} C[x := N] &\equiv \big((\lambda y C_1)C_2 \cdots C_n\big)[x := N] \\ &\to_\beta \big((C_1[y := C_2])C_3 \cdots C_n\big)[x := N]. \end{aligned}$$

And by induction,

$$\big((C_1[y := C_2])C_3 \cdots C_n\big)[x := N] \twoheadrightarrow c.$$

Thus we have a reduction $C[x := N] \twoheadrightarrow c$ as desired.

3. $C \equiv \delta C_1 \cdots C_n$. Then the reduction $C[x := M] \twoheadrightarrow_s c$ must contract the head δ by some rule $\theta : \delta_\theta\langle \vec{d}, \vec{y}\rangle \to P(\vec{y})$ (where each d_i is a ground constant). Accordingly, we rewrite C as

$$C \equiv \delta_\theta\langle \vec{D}, \vec{E}\rangle \vec{F}.$$

Then the reduction $C[x := M] \twoheadrightarrow_s c$ is of the form

$$\begin{aligned} C[x := M] &\equiv \delta_\theta\langle \vec{D}[x := M], \vec{E}[x := M]\rangle \vec{F}[x := M] \\ &\twoheadrightarrow \delta_\theta\langle \vec{d}, \vec{E}[x := M]\rangle \vec{F}[x := M] \\ &\to_\theta P(\vec{E}[x := M])\vec{F}[x := M] \\ &\twoheadrightarrow_s c, \end{aligned}$$

where each $D_i[x := M] \twoheadrightarrow_s d_i$ in turn. By Lemma 5.2,

$$P(\vec{E}[x := M]) \equiv P(\vec{E})[x := M],$$

so the reduction can be rewritten

$$\begin{aligned} C[x := M] &\equiv (\delta_\theta\langle \vec{D}, \vec{E}\rangle \vec{F})[x := M] \\ &\twoheadrightarrow (\delta_\theta\langle \vec{d}, \vec{E}\rangle \vec{F})[x := M] \\ &\to_\theta (P(\vec{E})\vec{F})[x := M] \\ &\twoheadrightarrow_s c. \end{aligned}$$

Again by Lemma 5.2,

$$P(\vec{E}[x := N])\vec{F}[x := N] \equiv (P(\vec{E})\vec{F})[x := N].$$

And by induction, $(P(\vec{E})\vec{F})[x := N] \twoheadrightarrow c$, and $D_i[x := N] \twoheadrightarrow d_i$. Thus we have found a reduction

$$\begin{aligned} C[x := N] &\equiv (\delta_\theta\langle \vec{D}, \vec{E}\rangle \vec{F})[x := N] \\ &\twoheadrightarrow (\delta_\theta\langle \vec{d}, \vec{E}\rangle \vec{F})[x := N] \\ &\to_\theta (P(\vec{E})\vec{F})[x := N] \\ &\twoheadrightarrow c. \end{aligned}$$

4. $C \equiv xC_1 \cdots C_n$. Then consider the term

$$C' \overset{\text{def}}{\equiv} MC_1 \cdots C_n.$$

Note that $C[x := M] \equiv C'[x := M]$, so $C'[x := M] \twoheadrightarrow_s c$. Moreover, C' must be of a form considered in the two previous cases, and so by the previous argument we conclude $C'[x := N] \twoheadrightarrow c$. Now consider the applicative *context*

$$C''[\cdot] \overset{\text{def}}{\equiv} [\cdot]C_1[x := N] \cdots C_n[x := N].$$

Since $C''[M] \equiv C'[x := N]$, we have $C''[M] \twoheadrightarrow c$. Finally, $M \sqsubseteq_{\text{app}} N$ implies $C''[N] \twoheadrightarrow c$; and

$$\begin{aligned} C''[N] &\equiv NC_1[x := N] \cdots C_n[x := N] \\ &\equiv C[x := N], \end{aligned}$$

so $C[x := N] \twoheadrightarrow c$.

Note that we need not consider the case $C \equiv yC_1 \cdots C_n$, where $y \not\equiv x$, since then $C[x := M]$ can never reduce to a ground constant. □

THEOREM 5.4 (Approximation Context Lemma). *In any PCF-like rewrite system,*

$$M \sqsubseteq_{\text{obs}} N \quad \textit{iff} \quad M \sqsubseteq_{\text{app}} N$$

for all closed terms M and N.

Proof.

($\Longrightarrow$) Trivial.

($\Longleftarrow$) It is sufficient to show the following: for all ground contexts $C[\cdot]$ and ground constants c, if $C[M] \twoheadrightarrow c$, then $C[N] \twoheadrightarrow c$.

Remember that the action of placing a term into the "holes" of a context differs from substitution only in that free variables of the term can be captured. But M and N are closed, with no free variables to capture; so for any context $C[\cdot]$,

$$\begin{array}{rrcl} & C[M] & \equiv & (C[x])[x := M], \\ \text{and} & C[N] & \equiv & (C[x])[x := N], \end{array}$$

where x is a fresh variable. So by Lemma 5.3, if $C[M] \twoheadrightarrow c$, then $C[N] \twoheadrightarrow c$ as well. □

From Corollary 3.3, we now immediately have the following.

THEOREM 5.5. *Every stable function model with Booleans that is adequate for a conservative extension of PCF defined by PCF-like rewrite rules is not equationally fully abstract.*

We remark that a simple sufficient condition to ensure that an extension of PCF by PCF-like rules is conservative is that δ-rules whose left-hand sides involve no new (non-PCF) constants must be exactly the rules of PCF.

Because we are unable to prove a Comparability Context Lemma for consistent PCF-like rewrite rules, Corollary 3.8 cannot be applied. Nevertheless, our analysis of comparability can be extended to show the following.

THEOREM 5.6. *Every bistable model with Booleans that is adequate for a conservative extension of PCF defined by consistent PCF-like rewrite rules is not equationally fully abstract.*

`tt,ff`	:	o	
$\underline{n}$	:	ι	for each integer $n \geq 0$
`succ,pred`	:	$\iota \rightarrow \iota$	
`zero?`	:	$\iota \rightarrow o$	
$\mathtt{cond}_o$	:	$o \rightarrow o \rightarrow o \rightarrow o$	
$\mathtt{cond}_\iota$	:	$o \rightarrow \iota \rightarrow \iota \rightarrow \iota$	
$\mathtt{Y}_\sigma$	:	$(\sigma \rightarrow \sigma) \rightarrow \sigma$	for each type σ

FIG. 1. *Constants of PCF.*

6. Conclusions and future work. We have extended the metatheory of term-rewriting semantics for simply typed λ-calculi and have shown that certain denotational models, in particular those based on stable and strongly stable domains, cannot be fully abstract for such operational semantics. Our proof exploits the lack of order-extensionality in these domains.

We conjecture that our methods and results will extend to untyped versions of PCF-like languages. Extensions to lazy and call-by-value languages also seem plausible, though with more difficulties, since higher-order terms now yield observations and the notion of lazy model is more technical.

The category of sequential algorithms [7] is technically not a model in our sense, but is like the stable model in that it is a Cartesian Closed Category with partially ordered function objects that are not pointwise ordered. We believe that with some minor modifications our results will apply to it as well.

Although we are able to show the failures of some order-extensional models, like the bistable models, the extensional embedding methods of [13] offer a more sophisticated way to restore order-extensionality which, for example, guarantees that the theory of the extensionally embedded models includes that of cpo's. We do not know whether these models can avoid the kind of failure of full abstraction that we have identified.

Finally, the work of Cartwright, Curien, and Felleisen raises some interesting issues [15]. They extend PCF with a higher-order operator, catch, that distinguishes functions based on their sequential evaluation: when applied to a functional expression, catch returns the index of the argument that the expression would evaluate first, if it were applied. This is enough, for example, to distinguish `True` and `True!`, since `True!` uses its (sole) argument first, while `True` does not use its argument at all.

The resulting language is fully abstract with an extension of the sequential algorithms model; however, it does not satisfy the Context Lemma (we still have `True!` $\sqsubseteq_{\text{app}}$ `True`, but no longer `True!` $\sqsubseteq_{\text{obs}}$ `True`). But by further adding "errors" to the language and model, they recover the Context Lemma and retain full abstraction.

There are two reasons that this does not contradict our results. First, although extensional, their language is not defined by PCF-like rules; whether it can be defined by such rules is an interesting open question. Second, extensionality is only achieved by adding "error" elements to the ground types of the model, and thus their model does not meet our technical definition of "model with Booleans."

Appendix A. PCF. Because we will work with both PCF and its extensions, we give the general definitions for simply typed λ-calculi. A language is parameterized by its ground types and typed constants; for instance, PCF's ground types are the Booleans o and the numerals ι, and its constants are listed in Figure 1.

$$\begin{aligned}
\texttt{cond}\,\texttt{tt}\;x\;y &\to x\\
\texttt{cond}\,\texttt{ff}\;x\;y &\to y\\[1ex]
\texttt{zero?}\,\underline{0} &\to \texttt{tt}\\
\texttt{zero?}\,\underline{n+1} &\to \texttt{ff}\\[1ex]
\texttt{succ}\,\underline{n} &\to \underline{n+1}\\[1ex]
\texttt{pred}\,\underline{0} &\to \underline{0}\\
\texttt{pred}\,\underline{n+1} &\to \underline{n}\\[1ex]
\texttt{Y}\,f &\to f(\texttt{Y}\,f)
\end{aligned}$$

FIG. 2. *Rewrite rules for PCF.*

The set of *types* of the language is the least set containing the ground types and $(\sigma \to \tau)$ for types σ and τ. The set of *first-order* types is the least set containing the ground types and $(\sigma \to \tau)$ for ground types σ and first-order types τ.

The typed *terms* of the language are defined inductively:

- A constant δ^σ is a term of type σ.
- A variable x^σ is a term of type σ.
- If M is a term of type $(\sigma \to \tau)$ and N is a term of type σ, then (MN) is a term of type τ.
- If M is a term of type τ, then $(\lambda x^\sigma M)$ is a term of type $(\sigma \to \tau)$.

We omit types and parentheses whenever possible, adopting the standard conventions of association: application associates to the left and "$\to$" associates to the right. We will use $M, N, P, \dots$ to denote arbitrary terms; $x, y, z, \dots$ to denote arbitrary variables; and $\sigma, \tau, \gamma, \dots$ to denote arbitrary types. δ will always denote a constant, and c will always be a ground constant. The binary relation symbol $\equiv$ denotes syntactic equality.

Free and bound variables are defined as usual, and we consider terms that are identical modulo a change of bound variables to be syntactically identical. A term is *closed* if it has no free variables; otherwise it is *open.* A *program* is a closed term of ground type.

A *substitution* is a type-respecting mapping of variables to terms. Substitutions are extended to terms as usual (taking care to avoid capture of free variables), and are written postfix, so that $M\rho$ is the application of the substitution ρ to the term M. We call $M\rho$ an *instance* of M. If $\vec{x} \equiv x_1, \dots, x_n$ and $\vec{N} \equiv N_1, \dots, N_n$, then $[\vec{x} := \vec{N}]$ is the substitution that maps each x_i to N_i (simultaneously) and is the identity otherwise. A special case is $[x := N]$, so that $M[x := N]$ is the result of substituting N for x in M. Sometimes we write $M \equiv M(\vec{x})$, with the intent that $M(\vec{N}) \equiv M[\vec{x} := \vec{N}]$.

A *context* $C[\cdot]$ is a term with some "holes." $C[M]$ denotes the result of putting M into the holes of $C[\cdot]$, which may cause free variables of M to become bound. We say $C[\cdot]$ is a *program context* for M if $C[M]$ is a closed term of ground type.

The interpreter of the language is defined via a rewrite system; any set of δ-rules, together with the classical rule (β), induces the one-step reduction relation $\to$. The relation $\twoheadrightarrow$ is the reflexive transitive closure of $\to$. Figure 2 gives the δ-rules for PCF.

Appendix B. Simply typed models. Here we develop the general framework for function-based models of simply typed λ-calculi.

A *type frame* $\{[\![\sigma]\!]\}$ is collection of sets indexed by type such that $[\![\sigma \to \tau]\!]$ is a set of functions from $[\![\sigma]\!]$ to $[\![\tau]\!]$. The sets $[\![\sigma]\!]$ are called *domains*, and the elements of each $[\![\sigma]\!]$ are called *meanings* or *values* of type σ.

Since our discussion focuses on issues of adequacy and full abstraction, we also require the following:

- there is a partial order $\sqsubseteq_\sigma$ associated with each domain $[\![\sigma]\!]$;
- the functions of $[\![\sigma \to \tau]\!]$ are monotone with respect to the orderings $\sqsubseteq_\sigma$ and $\sqsubseteq_\tau$; and
- the relation $\sqsubseteq_{\sigma\to\tau}$ refines the pointwise relation on functions $f, g \in [\![\sigma \to \tau]\!]$, i.e.,

$$f \sqsubseteq_{\sigma\to\tau} g \quad \text{implies} \quad f(d) \sqsubseteq_\tau g(d) \text{ for all } d \in [\![\sigma]\!].$$

The last two conditions say that function application is monotone in both arguments; this implies that models, defined below, are compositional.

An *environment* is a type-respecting mapping from variables to values. If ρ is an environment, then the environment $\rho[x := d]$ is ρ with the value of x updated to d:

$$\rho[x := d](y) = \begin{cases} d & \text{if } y \equiv x, \\ \rho(y) & \text{otherwise.} \end{cases}$$

An *interpretation* is a type-respecting mapping from constants to values. For a given type frame $\{[\![\sigma]\!]\}$ and interpretation $\mathcal{I}$, we can try to define a *model*, $[\![\cdot]\!]$, that is a mapping from each term to a meaning with respect to an environment, satisfying the following conditions:

$$[\![\delta]\!]\rho = \mathcal{I}(\delta), \tag{1}$$

$$[\![x]\!]\rho = \rho(x), \tag{2}$$

$$[\![(MN)]\!]\rho = ([\![M]\!]\rho)([\![N]\!]\rho), \tag{3}$$

$$([\![\lambda x M]\!]\rho)(d) = [\![M]\!]\rho[x := d]. \tag{4}$$

Implicit in condition (4) is the requirement that the function defined to be $([\![\lambda x M]\!]\rho)$ must be an element in an element of the type frame. In other words, a model is a type frame that is closed under lambda-definability. Such closure certainly does not hold for all type frames (cf. [26]).

The meaning of a closed term is the same in any environment:

$$[\![M]\!]\rho = [\![M]\!]\rho'$$

for all closed M and arbitrary ρ and ρ'. Therefore, we sometimes write $[\![M]\!]$ for the meaning of a closed term M, omitting the environment.

B.1. Continuity. We give the standard definitions for cpos and continuous functions, then define the cpo model of PCF.

A *partial order* or *poset* is a set D together with a binary relation $\sqsubseteq$ that is reflexive, transitive, and antisymmetric. We will refer to the partial order $\langle D, \sqsubseteq\rangle$ as just D. A subset $X \subseteq D$ is *directed* if every finite subset of X has an upper bound in X. A partial order D is a *complete partial order* or *cpo* if it has a least element $\bot_D$ and every directed subset $X \subseteq D$ has a least upper bound $\sqcup X$. We omit the subscript D in $\bot_D$ when it can be recovered from context. For any set X we define the cpo $X_\bot$, with elements $X \cup \{\bot_X\}$, ordered $x \sqsubseteq y$ iff $x = y$ or $x = \bot_X$.

A function $f : D \to E$ between posets is *monotone* if $f(x) \sqsubseteq_E f(y)$ whenever $x \sqsubseteq_D y$. We say f is *continuous* if it is monotone and $f(\sqcup X) = \sqcup f(X)$ for every directed $X \subseteq D$.

The set $D \to_c E$ of continuous functions from cpo D to cpo E is a cpo under the pointwise order $\sqsubseteq_{\mathrm{p}}$, defined as follows:

$$f \sqsubseteq_{\mathrm{p}} g \quad \text{iff} \quad f(x) \sqsubseteq_E g(x) \text{ for all } x \in D.$$

If D is a cpo and $f : D \to D$ is continuous, then f has a least fixed point $\mathit{fix}(f)$. The function *fix* itself is continuous, which will allow us to interpret the recursion operator Y.

Now we define the cpo model $\mathcal{C}[\![\cdot]\!]$ of PCF, based on continuous functions and cpos. First we construct a type frame with ground domains $\mathcal{C}[\![o]\!] = \{\mathit{tt}, \mathit{ff}\}_\bot$ and $\mathcal{C}[\![\iota]\!] = \{0, 1, 2, \ldots\}_\bot$, and higher-order domains $\mathcal{C}[\![\sigma \to \tau]\!] = \mathcal{C}[\![\sigma]\!] \to_c \mathcal{C}[\![\tau]\!]$. The cpo model of PCF is then the model $\mathcal{C}[\![\cdot]\!]$ associated with $\{\mathcal{C}[\![\sigma]\!]\}$ and the *standard interpretation*: the ground constants are interpreted in the obvious way; the constants Y_σ are interpreted as *least* fixed-point operators; and the interpretation of the remaining function constants is determined by the condition that the rewrite rules of Figure 2 be valid as equations.

THEOREM B.1 (Plotkin [31], Sazonov [32]). *The cpo model $\mathcal{C}[\![\cdot]\!]$ is adequate but not fully abstract for PCF.*

B.2. Stability. If D is a partial order and $X \subseteq D$, then X is *bounded* or *consistent* if there is an element $y \in D$ such that $x \sqsubseteq y$ for all $x \in X$. If elements x and y are consistent, we will write $x \uparrow y$. We say D is *bounded complete* if every bounded subset $X \subseteq D$ has a least upper bound $\sqcup X$.

An element $a \in D$ is *compact* if, for every directed $X \subseteq D$ with $a \sqsubseteq \sqcup X$, there is some $x \in X$ such that $a \sqsubseteq x$. We define $\mathbf{K}D$, the *kernel* of D, to be the set of compact elements of D. The cpo D is *algebraic* if, for every $x \in D$, the set $\downarrow x = \{ a \in \mathbf{K}D \mid a \sqsubseteq x \}$ is directed and $\sqcup \downarrow x = x$.

The greatest lower bound of a set X is denoted $\sqcap X$. A cpo is *distributive* if $x \sqcap (y \sqcup z) = (x \sqcap y) \sqcup (x \sqcap z)$ whenever y and z are consistent. An algebraic cpo D *has property I* if $\downarrow a$ is finite for each $a \in \mathbf{K}D$. A *dI-domain* is a distributive, bounded complete cpo that has property I.

A continuous function f between dI-domains is *stable* if whenever $x \uparrow y$, we have that $f(x \sqcap y) = f(x) \sqcap f(y)$. We let $D \to_s E$ be the set of stable functions between dI-domains D and E. As noted in [6], $D \to_s E$ ordered pointwise is *not* a dI-domain; accordingly, we define the stable ordering $\sqsubseteq_{\mathrm{s}}$:

$$f \sqsubseteq_{\mathrm{s}} g \qquad \text{iff} \qquad f(x) = f(y) \sqcap g(x) \text{ whenever } x \sqsubseteq y.$$

If D and E are dI-domains, then $D \to_s E$ is a dI-domain under the stable order.

It must be noted that the stable order is quite different from the pointwise order. For instance, consider the monotone Boolean functions, listed in Figure 3. These functions are both continuous and stable, and so they are elements of both the continuous and stable type frames. However, the stable ordering of $o \to o$ (Figure 5) is different from its pointwise ordering (Figure 4). In particular, consider *True*, the constant *tt* function, and *True!*, the strict constant *tt* function. Although *True!* $\sqsubseteq_{\mathrm{p}}$ *True*, we have *True!* $\not\sqsubseteq_{\mathrm{s}}$ *True*, since $\bot \sqsubseteq_{\mathrm{s}} \mathit{tt}$ but

$$\mathit{True!}(\bot) = \bot \quad \neq \quad \mathit{tt} = (\mathit{True!}(\mathit{tt}) \sqcap \mathit{True}(\bot)).$$

(It is this that permits the existence of the function *truesep* that was needed in Corollary 3.3.)

Nevertheless, a stable model $\mathcal{S}[\![\cdot]\!]$ of PCF, based on dI-domains and stable functions, can be defined in much the same way as the cpo model. The ground domains $\mathcal{S}[\![o]\!]$ and $\mathcal{S}[\![\iota]\!]$ of the stable type frame are identical to the ground domains of the cpo model. At higher types, however, we use stable functions: $\mathcal{S}[\![\sigma \to \tau]\!] = \mathcal{S}[\![\sigma]\!] \to_s \mathcal{S}[\![\tau]\!]$. Then we let $\mathcal{S}[\![\cdot]\!]$ be the model associated with the stable type frame and the (stable) standard interpretation (cf. the interpretation of the cpo model).

THEOREM B.2 (Berry [6]). *The stable model $\mathcal{S}[\![\cdot]\!]$ is adequate but not fully abstract for PCF.*

Appendix C. Standard reductions in PCF-like rewrite systems.

C.1. Preliminaries. This appendix gives a full definition of standard reductions and proof of the Standardization Theorem. In this section, we sketch out some of the basic terminology of rewriting systems. Section C.2 introduces descendants, which allow us to trace subterms from step to step in a reduction. In §C.3, we show that a very weak form of confluence holds for PCF-like systems; this property will be essential in proving the Standardization Theorem. Section C.4 introduces labelled rewrite systems and proves that they are strongly normalizing. The labelled systems will be used in the proof of Standardization. The standard reductions are defined in §C.5, and Standardization is proved in §C.6. The proof is a variation of Klop's proof for the pure λ-calculus [24] and involves a rewriting system on reductions. The system successively rewrites nonstandard reduction paths to "more standard" paths; Standardization is proved by showing that the system is strongly normalizing, and that normal forms are standard reductions.

Our presentation of the machinery used to state and prove Standardization is necessarily brief. Much of the material is covered in more depth in standard references [4, 24]. Throughout, we will work with a PCF-like rewrite system given by a language, $\mathcal{L}$, and set, Θ, of linear ground δ-rules.

We assume that the reader is familiar with the following terminology. The notation $M \subset N$ denotes that M is a *subterm* of N. A subterm may appear several times in a term; multiple *occurrences* of a subterm can be distinguished by their *paths*, which specify the exact position of a subterm inside the term. When we speak of a subterm $M \subset N$ we implicitly mean a particular occurrence of M in N; the disambiguating paths are omitted.

Note that $M \to N$ iff there is an instance $\Delta \to \Delta'$ of a rule π such that $\Delta \subset M$, and N is obtained from M by replacing Δ with Δ'. We will write $M \xrightarrow{\Delta}_{\pi} N$ in this case, and we call Δ a *(π)-redex* and Δ' its *(π)-contractum.*

A *reduction (path)* $\boldsymbol{\sigma}$ is a sequence

$$\boldsymbol{\sigma} : M_1 \xrightarrow{\Delta_1}_{\pi_1} M_2 \xrightarrow{\Delta_2}_{\pi_2} M_3 \xrightarrow{\Delta_3}_{\pi_3} \cdots.$$

We will use $\boldsymbol{\sigma}, \boldsymbol{\tau}, \ldots$ to refer to reduction paths. Two reductions are *coinitial* if they start in the same term and *cofinal* if they end in the same term.

C.2. Descendants. Consider some possible effects of a reduction $M \to N$ on a subterm $\Delta \subset M$:

- Δ could be erased, as in $(\lambda x.y)\Delta \to y$.
- Δ could be copied to some instances in N, as in $(\lambda x.\delta xx)\Delta \to \delta\Delta\Delta$.
- Δ could be left untouched and in its original position, as in $\Delta((\lambda x.x)y) \to \Delta y$.

Function	tt	ff	$\bot$
True	tt	tt	tt
False	ff	ff	ff
True!	tt	tt	$\bot$
False!	ff	ff	$\bot$
Id	tt	ff	$\bot$
Not	ff	tt	$\bot$
$(tt \Rightarrow tt)$	tt	$\bot$	$\bot$
$(tt \Rightarrow ff)$	ff	$\bot$	$\bot$
$(ff \Rightarrow tt)$	$\bot$	tt	$\bot$
$(ff \Rightarrow ff)$	$\bot$	ff	$\bot$
Bot	$\bot$	$\bot$	$\bot$

FIG. 3. *Boolean functions.*

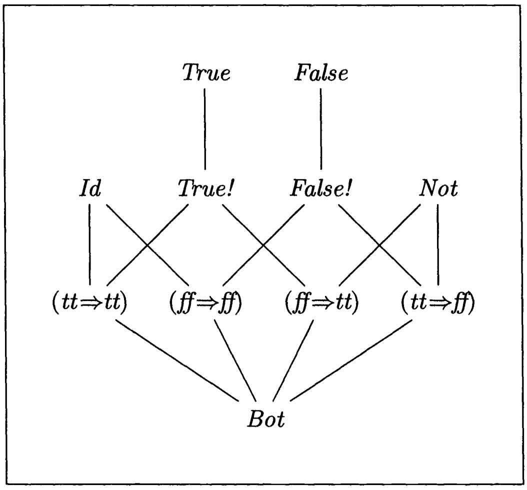


FIG. 4. *Pointwise ordering of $o \rightarrow o$.*

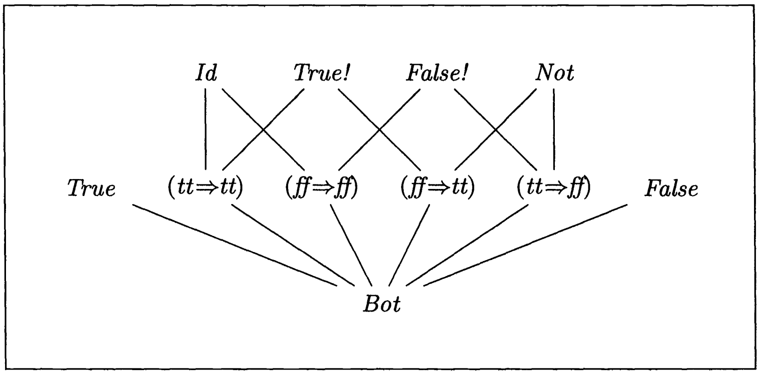


FIG. 5. *Stable ordering of $o \rightarrow o$.*

- The contracted redex might occur within Δ, transforming it into a syntactically different subterm in the same position.

In order to define and prove standardization, we will need to speak precisely about these cases, so we introduce *descendants*, which let us track a subterm throughout a reduction. We will not define descendants in their full generality, but only for certain subterms of interest. Our definition is equivalent to the standard definition [24] on those subterms.

Descendants are introduced via an annotated rewrite system derived from $\mathcal{L}$ and Θ, in which some λ's and δ's are marked with a $*$. Thus we define the language $\mathcal{L}_*$, whose symbols are those of $\mathcal{L}$, with the addition of λ_*, and δ_*^σ for each constant δ^σ of $\mathcal{L}$. The terms of $\mathcal{L}_*$ are defined inductively:

- A constant δ^σ or δ_*^σ is a term of type σ.
- A variable x^σ is a term of type σ.
- If M is a term of type $(\sigma \to \tau)$ and N is a term of type σ, then (MN) is a term of type τ.
- If M is a term of type τ, then $(\lambda x^\sigma M)$ and $(\lambda_* x^\sigma M)$ are terms of type $(\sigma \to \tau)$.

The *erasure* $|M| \in \mathcal{L}$ of $M \in \mathcal{L}_*$ is obtained from M by leaving out the $*$'s. Substitution for the language is defined in the obvious way (with λ_*'s binding variables just as λ's). The rules of the new system include β and the rule scheme β_*:

$$\beta_* : (\lambda_* x M)N \to M[x := N].$$

Similarly, the δ-rules Θ_* of the system are derived from the rules Θ. If θ is a rule of Θ,

$$\theta : \delta\langle \vec{c}, \vec{x} \rangle \to P(\vec{x}),$$

then Θ_* contains all rules of the form θ' and θ_*:

$$\theta' : \delta\langle \vec{c}', \vec{x} \rangle \to P(\vec{x}),$$
$$\theta_* : \delta_*\langle \vec{c}', \vec{x} \rangle \to P(\vec{x}),$$

where $\vec{c}'$ is any vector of $\mathcal{L}_*$ ground constants such that $|\vec{c}'| \equiv \vec{c}$.

There is a strong connection between the systems. Any Θ_*-reduction path $\boldsymbol{\sigma}$,

$$\boldsymbol{\sigma} : M_1 \overset{\Delta_1}{\longrightarrow}_{\pi_1} M_2 \overset{\Delta_2}{\longrightarrow}_{\pi_2} M_3 \overset{\Delta_3}{\longrightarrow}_{\pi_3} \cdots,$$

projects to a Θ-reduction path $|\boldsymbol{\sigma}|$:

$$|\boldsymbol{\sigma}| : |M_1| \overset{|\Delta_1|}{\longrightarrow}_{|\pi_1|} |M_2| \overset{|\Delta_2|}{\longrightarrow}_{|\pi_2|} |M_3| \overset{|\Delta_3|}{\longrightarrow}_{|\pi_3|} \cdots.$$

Conversely, for any $M \in \mathcal{L}_*$ and Θ-reduction path $\boldsymbol{\sigma} : |M| \to \cdots$, there is a unique *lift* of $\boldsymbol{\sigma}$ to a Θ_*-reduction path $\boldsymbol{\sigma}' : M \to \cdots$ such that $\boldsymbol{\sigma} \equiv |\boldsymbol{\sigma}'|$.

We will be interested in tracing subterms of the form $(\lambda x.M_1)M_2$ or $\delta M_1 \cdots M_n$ throughout a reduction; that is, β-redexes and possible δ-redexes. Accordingly, we introduce the following terminology. A subterm $(\lambda x.M_1)M_2$ or $\delta M_1 \cdots M_n$ of M is called a *predescendant of* M. If $\mathcal{F}$ is a set of predescendants of $M \in \mathcal{L}$, we write $(M, \mathcal{F})$ for the $\mathcal{L}_*$ term derived from M by marking the head λ or δ of each predescendant in $\mathcal{F}$ with a $*$.

DEFINITION C.1. *Suppose* $\boldsymbol{\sigma} : M \to \cdots \to N$ *is a* Θ*-reduction path.*

(i) *If Δ is a predescendant of M, its set of* descendants *in N relative to $\boldsymbol{\sigma}$, written $(\Delta/\boldsymbol{\sigma})$, is defined as follows.*
Let $M' \equiv (M, \{\Delta\})$ and lift $\boldsymbol{\sigma}$ to $\boldsymbol{\sigma}' : M' \to \cdots \to N'$. If $\Delta \equiv (\lambda x M_1) M_2$ (resp., $\Delta \equiv \delta M_1 \cdots M_n$), then $(\Delta/\boldsymbol{\sigma}) \overset{\text{def}}{=} \mathcal{F}$, where $\mathcal{F}$ is the unique set of subterms of N of the form $(\lambda x M_1') M_2'$ (resp., $\delta M_1' \cdots M_n'$), such that $N' \equiv (N, \mathcal{F})$.

(ii) *If $\mathcal{F}$ is a set of predescendants of M, its descendants $\mathcal{F}/\boldsymbol{\sigma}$ are defined*

$$\mathcal{F}/\boldsymbol{\sigma} \overset{\text{def}}{=} \bigcup \{ \Delta/\boldsymbol{\sigma} \mid \Delta \in \mathcal{F} \}.$$

(iii) *$\Delta \subset M$ is an* ancestor *of $\Delta' \subset N$ if $\Delta' \in \Delta/\boldsymbol{\sigma}$.*

For a given reduction $M_1 \to M_2 \to M_3 \to \cdots$, we will sometimes speak of descendants and ancestors for subterms of terms M_i and M_j, where i and j are any indices such that $j \geq i$. We do not specify the reduction from M_i to M_j, as it can be recovered from context.

NOTE C.2.

(i) *If $M \xrightarrow{\Delta} N$, then Δ has no descendants in N.*

(ii) *If $M \xrightarrow{\Delta}_\theta N$, where $\Delta \equiv \delta\langle \vec{c}, \vec{B}\rangle$, then no c_i has a descendant in N.*

We mention that the following important property holds for our PCF-like systems, since it does not hold for all rewrite systems [24].

NOTE C.3. *If $\Delta \subset M$ and $M \to N$, then descendants of Δ in N are disjoint.* Disjointness of descendants does not extend to $\twoheadrightarrow$, as we indicate here:

$$\begin{aligned}(\lambda y.(\lambda x.yx)y)(\lambda z.\delta_* z) &\to_\beta (\lambda x.(\lambda z.\delta_* z)x)(\lambda z.\delta_* z)\\ &\to_\beta (\lambda x.\delta_* x)(\lambda z.\delta_* z)\\ &\to_\beta \delta_*(\lambda z.\delta_* z).\end{aligned}$$

DEFINITION C.4. *Suppose M_i is a term in a reduction $\boldsymbol{\sigma}$,*

$$\boldsymbol{\sigma} : M_1 \xrightarrow{\Delta_1}_{\pi_1} M_2 \xrightarrow{\Delta_2}_{\pi_2} M_3 \xrightarrow{\Delta_3}_{\pi_3} \cdots.$$

(i) *We say $\Delta \subset M_i$ is* (π)-contracted (in $\boldsymbol{\sigma}$) *if for some $j \geq i$, Δ_j is a descendant of Δ and $\pi_j = \pi$.*

(ii) *We say $\Delta \subset M_i$ is* active (in $\boldsymbol{\sigma}$) *if there is a $\Delta' \subset \Delta$ that is contracted in $\boldsymbol{\sigma}$.*

Sometimes it will be useful to specify a set of subterms of some term M, and consider reductions from M in which only those subterms are contracted. Such reductions are called *developments*. Because we work with systems in which a subterm can contract by more than one rule, our definition of developments extends the standard definition by specifying a rule for each redex contracted in a development.

DEFINITION C.5. *Suppose the following: $\boldsymbol{\sigma}$ is a reduction from M to N; $\mathcal{F}$ is a set of subterms of M; and Π is a mapping that takes each $\Delta \in \mathcal{F}$ to a rule π_Δ.*

(i) *We call $\boldsymbol{\sigma}$ a* development *of $\mathcal{F}$ from M by Π, written $\boldsymbol{\sigma} : (M, \mathcal{F}) \overset{\Pi}{\twoheadrightarrow} N$, if each redex Δ' contracted in $\boldsymbol{\sigma}$ is a descendant of some $\Delta \in \mathcal{F}$, and Δ' is contracted by rule π_Δ.*

(ii) *We say a development $\boldsymbol{\sigma}$ is a* complete development, *written $\boldsymbol{\sigma} : (M, \mathcal{F}) \underset{\text{cpl}}{\overset{\Pi}{\twoheadrightarrow}} N$, if $\mathcal{F}/\boldsymbol{\sigma} = \emptyset$.*

When Π is evident from context, we will omit mention of it.

NOTE C.6. *If $\mathcal{F}$ is a set of n disjoint redexes of M, then clearly all complete developments of $\mathcal{F}$ from M are of length n and are cofinal.*

C.3. Properties related to confluence. Note C.6 is a special case of a much stronger theorem, the Finite Developments Theorem. We will not need to prove the Finite Developments Theorem in its full generality; this section proves a weaker result that will be sufficient for our application.

DEFINITION C.7. *We say two δ-redexes Δ_1 and Δ_2* overlap *if either*

(i) *they share the same head δ, or*

(ii) *one Δ_i appears as a critical argument of the other.*

Note that in case (ii), *the Δ_i must be a ground constant.*

Often, rewrite systems are constrained to avoid overlapping redexes; such systems are guaranteed to be confluent. Because we allow overlapping rules, our systems are not confluent in general. However, they do satisfy the following much weaker property, which will be essential in our proof of standardization.

LEMMA C.8. *Suppose $\boldsymbol{\sigma}_1 : M_0 \overset{\Delta_1}{\twoheadrightarrow} M_1$ and $\boldsymbol{\sigma}_2 : M_0 \overset{\Delta_2}{\twoheadrightarrow} M_2$, where Δ_1 and Δ_2 do not overlap. Then complete developments of $\Delta_2/\boldsymbol{\sigma}_1$ from M_1 and $\Delta_1/\boldsymbol{\sigma}_2$ from M_2 are finite and cofinal.*

Proof. For each of the various cases on the relative positions of Δ_1 and Δ_2 in M_0, we find a term M_3 that is the final term of every complete development of $\Delta_1/\boldsymbol{\sigma}_2$ and $\Delta_2/\boldsymbol{\sigma}_1$:

$$\begin{tikzcd}
M_0 \arrow[r, "\Delta_1"] \arrow[d, "\Delta_2"] & M_1 \arrow[d, twoheadrightarrow, "\Delta_2/\sigma_1."] \\
M_2 \arrow[r, twoheadrightarrow, "\Delta_1/\sigma_2"] & M_3
\end{tikzcd}$$

1. Δ_1 and Δ_2 are disjoint. Then M_0, M_1, and M_2 can be written

$$M_0 \equiv \cdots \Delta_1 \cdots \Delta_2 \cdots,$$
$$M_1 \equiv \cdots \Delta_1' \cdots \Delta_2 \cdots,$$
$$M_2 \equiv \cdots \Delta_1 \cdots \Delta_2' \cdots,$$

where Δ_1' and Δ_2' are the respective contractums of Δ_1 and Δ_2. Now defining

$$M_3 \overset{\text{def}}{\equiv} \cdots \Delta_1' \cdots \Delta_2' \cdots,$$

we see that the only complete development of $\Delta_2/\boldsymbol{\sigma}_1$ is $M_1 \overset{\Delta_2}{\twoheadrightarrow} M_3$, and the only complete development of $\Delta_1/\boldsymbol{\sigma}_2$ is $M_2 \overset{\Delta_1}{\twoheadrightarrow} M_3$, as desired.

2. $\Delta_1 \subset \Delta_2$. Then there is a unique descendant Δ_2' of Δ_2 in M_1, and we consider three subcases.

(a) $\Delta_2 \equiv (\lambda x. \cdots \Delta_1 \cdots)N$. Then we can write M_0, M_1, and M_2 as

$$M_0 \equiv \cdots ((\lambda x. \cdots \Delta_1 \cdots)N) \cdots,$$
$$M_1 \equiv \cdots ((\lambda x. \cdots \Delta_1' \cdots)N) \cdots,$$
$$M_2 \equiv \cdots ((\cdots \Delta_1 \cdots)[x := N]) \cdots,$$

where Δ_1' is the contractum of Δ_1, and $\Delta_2' \equiv (\lambda x. \cdots \Delta_1' \cdots)N$. If we take

$$M_3 \overset{\text{def}}{\equiv} \cdots ((\cdots \Delta_1' \cdots)[x := N]) \cdots,$$

then the only complete development of $\Delta_2/\boldsymbol{\sigma}_1$ is $M_1 \xrightarrow{\Delta_2'}_\beta M_3$. Furthermore, substitutivity holds for PCF-like rewrite systems; that is,

$$M \xrightarrow{\Delta} M' \Longrightarrow M[x := N] \xrightarrow{\Delta'} M'[x := N],$$

where Δ' is Δ with any free occurrences of x replaced by N. Thus the only complete development of $\Delta_1/\boldsymbol{\sigma}_2$ is $M_2 \to M_3$.

(b) $\Delta_2 \equiv (\lambda x.N)(\cdots \Delta_1 \cdots)$. Then M_0, M_1, and M_2 can be written

$$\begin{aligned} M_0 &\equiv \cdots ((\lambda x.N)(\cdots \Delta_1 \cdots)) \cdots, \\ M_1 &\equiv \cdots ((\lambda x.N)(\cdots \Delta_1' \cdots)) \cdots, \\ M_2 &\equiv \cdots (N[x := (\cdots \Delta_1 \cdots)]) \cdots, \end{aligned}$$

where Δ_1' is the contractum of Δ_1, and $\Delta_2' \equiv (\lambda x.N)(\cdots \Delta_1' \cdots)$. Defining

$$M_3 \overset{\text{def}}{\equiv} \cdots (N[x := (\cdots \Delta_1' \cdots)]) \cdots,$$

we see that the only complete development of $\Delta_2/\boldsymbol{\sigma}_1$ is $M_1 \xrightarrow{\Delta_2'}_\beta M_3$. Furthermore, descendants of Δ_1 in M_2 are disjoint, and any contraction of them in turn is a reduction $M_2 \xrightarrow{\Delta_1} \cdots \xrightarrow{\Delta_1} M_3$.

(c) $\Delta_2 \equiv \delta_\theta\langle \cdots, \cdots (\cdots \Delta_1 \cdots) \cdots \rangle$. Then we write M_0, M_1, and M_2 as

$$\begin{aligned} M_0 &\equiv \cdots (\delta_\theta\langle \cdots, \cdots (\cdots \Delta_1 \cdots) \cdots \rangle) \cdots, \\ M_1 &\equiv \cdots (\delta_\theta\langle \cdots, \cdots (\cdots \Delta_1' \cdots) \cdots \rangle) \cdots, \\ M_2 &\equiv \cdots (P_\theta(\cdots (\cdots \Delta_1 \cdots) \cdots)) \cdots, \end{aligned}$$

where Δ_1' is the contractum of Δ_1, and $\Delta_2' \equiv \delta_\theta\langle \cdots, \cdots (\cdots \Delta_1' \cdots) \cdots \rangle$. Defining

$$M_3 \overset{\text{def}}{\equiv} \cdots (P_\theta(\cdots (\cdots \Delta_1' \cdots) \cdots)) \cdots,$$

we see that the only complete development of $\Delta_2/\boldsymbol{\sigma}_1$ is $M_1 \xrightarrow{\Delta_2'}_\theta M_3$. And just as in case 2(b), the descendants of Δ_1 in M_2 are disjoint, so by contracting them in turn we find a reduction $M_2 \xrightarrow{\Delta_1} \cdots \xrightarrow{\Delta_1} M_3$.

3. $\Delta_2 \subset \Delta_1$. This case is handled exactly as case 2. □

C.4. A labelled λ-calculus. For any PCF-like rewrite system, there is a corresponding *labelled* PCF-like system that is strongly normalizing. The labelling technique has led to some of the simplest proofs for many syntactic properties, and we will use it in our proof of standardization. This section introduces labelled calculi and proves that they are strongly normalizing.

The labelled system is similar to the system that we introduced earlier to define descendants. However, the systems are also different in important ways, since they are intended for different purposes. In the labelled system, we will mark δ's with nonnegative integers instead of $*$'s, and we will not need to mark λ's. Furthermore, we do not allow unmarked δ's. The reasons for this will become apparent in what follows.

For any PCF-like language $\mathcal{L}$, the language $\mathcal{L}_\mathbb{N}$ is just the PCF-like language with constants δ_n^σ for each constant δ^σ of $\mathcal{L}$ and each $n \in \mathbb{N}$.

NOTATION C.9.

(i) *If $M \in \mathcal{L}_{\mathbb{N}}$, then $|M| \in \mathcal{L}$ is the term derived from M by erasing the labels on the constants.*

(ii) *If $M \in \mathcal{L}$, then $M^n \in \mathcal{L}_{\mathbb{N}}$ is the term derived from M by labelling each constant with n.*

The δ-rules $\Theta_{\mathbb{N}}$ of the labelled calculus are defined as follows. If θ is a rule of Θ,

$$\theta : \delta\langle \vec{c}, \vec{x}\rangle \to P(\vec{x}),$$

then $\Theta_{\mathbb{N}}$ contains all rules of the form $\theta_{\mathbb{N}}$:

$$\theta_{\mathbb{N}} : \delta_{n+1}\langle \vec{c}', \vec{x}\rangle \to P^n(\vec{x}),$$

where $\vec{c}'$ is a vector of $\mathcal{L}_{\mathbb{N}}$ ground constants such that $|\vec{c}'| \equiv \vec{c}$. Note that there is no rule for any δ_0.

The projection $|\boldsymbol{\sigma}|$ of a $\Theta_{\mathbb{N}}$-reduction path $\boldsymbol{\sigma}$ is defined in the obvious way. And any finite Θ-reduction $\boldsymbol{\sigma}$ can be lifted to a $\Theta_{\mathbb{N}}$-reduction $\boldsymbol{\sigma}'$ such that $\boldsymbol{\sigma} \equiv |\boldsymbol{\sigma}'|$ (e.g., label each constant in the first term of $\boldsymbol{\sigma}$ by the length of $\boldsymbol{\sigma}$).

DEFINITION C.10. *A term M is* strongly normalizable (SN) *if all reductions starting at M are finite.*

THEOREM C.11 (Strong Normalization). *Every $\mathcal{L}_{\mathbb{N}}$ term is strongly normalizable.*

The rest of this section lays out the proof of strong normalization. We use a straightforward extension of the method of [18].

DEFINITION C.12. *The notion of* strong computability (SC) *of a term is defined by induction as follows:*

(i) *A term of ground type is SC iff it is SN.*

(ii) *A term $M^{(\sigma\to\tau)}$ is SC iff, for every SC term N^σ, the term $(MN)^\tau$ is SC.*

NOTE C.13. *By Definition* C.12(ii), *a term M is SC iff, for all vectors $\vec{N}$ of SC terms driving M to ground type, the term $M\vec{N}$ is SC. And by Definition* C.12(i), *such an $M\vec{N}$ is SC iff it is SN.*

DEFINITION C.14. *An* atom *is a variable or a constant δ_n with no rule.*

LEMMA C.15.

(i) *If a is an atom and $\vec{N}$ is a vector of SN terms, then the term $a\vec{N}$ is SC.*

(ii) *Every SC term M is SN.*

Proof. This is a proof by induction on the type of $a\vec{N}$ and M.

1. Basis: $a\vec{N}$ and M have ground type.
   (i) Since each N_i is SN, $a\vec{N}$ must be SN, and therefore SC by Definition C.12(i).
   (ii) This follows by Definition C.12(i).
2. Induction: $a\vec{N}$ and M have type $\sigma \to \tau$.
   (i) Let P^σ be SC. By the induction hypothesis (ii), P is SN. Then by induction, the term $(a\vec{N}P)^\tau$ is SC. Therefore, so is $a\vec{N}$ by Definition C.12(ii).
   (ii) Let x^σ be a variable not occurring in M. By the induction hypothesis (i), x is SC. Then $(Mx)^\tau$ is SC, and therefore SN by induction. But any subterm of an SN term is SN, so M is SN as well. □

LEMMA C.16. *If N is SC and $M[x := N]$ is SC, then so is $(\lambda x M)N$.*

Proof. Let $\vec{P} \equiv P_1, \ldots, P_n$ be a vector of SC terms driving M to ground type. Since $M[x := N]$ is SC, the term

$$(M[x := N])\vec{P} \tag{5}$$

is SN by Note C.13. The lemma follows from Note C.13 if we can prove that

$$(\lambda x M)N\vec{P} \tag{6}$$

is SN.

Now since (5) is SN, all of its subterms are SN, including $M[x := N], \vec{P}$. Furthermore, by hypothesis and the preceding lemma, N is SN. Therefore, an infinite reduction from (6) cannot consist entirely of contractions in $M, N, P_1, \ldots, P_n$. So an infinite reduction of (6) must have the form

$$\begin{aligned}(\lambda x M)NP_1 \cdots P_n &\twoheadrightarrow (\lambda x M')N'P_1' \cdots P_n' \\ &\rightarrow M'[x := N']P_1' \cdots P_n' \\ &\twoheadrightarrow \cdots\end{aligned}$$

(where $M \twoheadrightarrow M'$, etc.) From the reductions $M \twoheadrightarrow M'$ and $N \twoheadrightarrow N'$, we have

$$M[x := N] \twoheadrightarrow M'[x := N']$$

Then we can construct an infinite reduction from (5) as follows:

$$\begin{aligned}M[x := N]P_1 \cdots P_n &\twoheadrightarrow M'[x := N']P_1' \cdots P_n' \\ &\twoheadrightarrow \cdots .\end{aligned}$$

But this contradicts the fact that (5) is SN. Therefore there is no infinite reduction from (6); it must be SN. □

LEMMA C.17. *Consider a constant δ and a vector $\vec{N}$ of SC terms driving δ to ground type. If for each rule θ on δ,*

$$\theta : \delta_\theta\langle \vec{c}, \vec{x}\rangle \rightarrow P_\theta(\vec{x}),$$

where $\delta\vec{N} \equiv \delta_\theta\langle \vec{N_1}, \vec{N_2}\rangle\vec{N_3}$, we have that

$$P_\theta(\vec{N_2})\vec{N_3} \tag{7}$$

is SC, then $\delta\vec{N}$ is SC.

Proof. We must show that $\delta\vec{N}$ is SN. Since the $\vec{N}$ are SC, by Lemma C.15 they are SN. Therefore, any infinite reduction from $\delta\vec{N}$ must look like

$$\begin{aligned}\delta_\theta\langle \vec{N_1}, \vec{N_2}\rangle\vec{N_3} &\twoheadrightarrow \delta_\theta\langle \vec{c}, \vec{N_2}'\rangle\vec{N_3}' \\ &\rightarrow P_\theta(\vec{N_2}')\vec{N_3}' \\ &\twoheadrightarrow \cdots,\end{aligned}$$

where $\vec{N_1} \twoheadrightarrow \vec{c}$, $\vec{N_2} \twoheadrightarrow \vec{N_2}'$, etc. But then we can construct an infinite reduction from (7) as follows:

$$\begin{aligned}P_\theta(\vec{N_2})\vec{N_3} &\twoheadrightarrow P_\theta(\vec{N_2}')\vec{N_3}' \\ &\twoheadrightarrow \cdots .\end{aligned}$$

But as (7) is SC, by Lemma C.15 it is SN, a contradiction. Therefore, $\delta\vec{N}$ is SN. □

LEMMA C.18. *For any term M and substitution $\rho \equiv [\vec{x} := \vec{N}]$, where each N_i is SC, the term $M\rho$ is SC.*

Proof. The proof is by induction on the lexicographic ordering of (m, M), where m is the maximum δ-index appearing in M.

1. M is a variable x_i. Then $M\rho$ is N_i and the result follows.
2. M is an atom distinct from $x_1, \ldots, x_n$. Then $M\rho \equiv M$ which is SC by Lemma C.15. Note that this includes all constants δ_0.
3. $M \equiv \delta_{m+1}$. Then $M\rho \equiv \delta_{m+1}$. Thus it is sufficient to show that for any vector $\vec{N}'$ of SC terms driving δ_{m+1} to ground type, the term $\delta_{m+1}\vec{N}'$ is SC. Consider any rule θ on δ_{m+1}:

$$\theta : \delta_{m+1}\langle\vec{c}, \vec{x}\rangle \to P(\vec{x}).$$

   By construction of the labelled calculus, no constants in P are labelled with an index greater than m. Thus we can apply the induction hypothesis to P. If we rewrite $\delta_{m+1}\vec{N}'$ as $\delta_{m+1}\langle\vec{N_1}', \vec{N_2}'\rangle\vec{N_3}'$, by induction $P(\vec{N_2}')$ is SC. Then by the definition of SC, the term $P(\vec{N_2}')\vec{N_3}'$ is SC. Therefore, by Lemma C.17, $\delta_{m+1}\vec{N}'$ is SC.
4. $M \equiv \lambda y^\sigma M_1$. Then $M\rho \equiv \lambda y(M_1\rho)$, neglecting changes in bound variables. To show that $M\rho$ is SC we must show that for all SC terms N^σ, the term $(M\rho)N$ is SC. But $(M\rho)N \equiv (\lambda y(M_1\rho))N$, and

$$(M_1\rho)[y := N] \equiv M_1[x_1 := N_1] \cdots [x_n := N_n][y := N],$$

   which is SC by induction. Therefore $(\lambda y(M_1\rho))N$ is SC by Lemma C.16.
5. $M \equiv M_1M_2$. Then $M\rho \equiv (M_1\rho)(M_2\rho)$, and $M_1\rho$ and $M_2\rho$ are SC by induction. Therefore, $M\rho$ is SC by Definition C.12(ii). □

Proof of Theorem C.11 *(Strong Normalization).* By Lemma C.18, every term M is SC (just let $\vec{x}$ be empty). Then by Lemma C.15, M is strongly normalizing. □

C.5. Standard reductions. Our definition of standard reductions is similar to that of [20], with a few important differences. The "linear ground" restriction imposed on our systems gives us a particularly simple class of rewrite rules, and this simplicity carries over to the definition of standard reductions. On the other hand, the systems of [20] do not include λ-abstraction and forbid overlapping rewrite rules, which we allow.

Overlapping rules do not add much complication to the definition of standard reductions, but they are more of an obstacle in the proof of standardization. Overlapping systems are not confluent in general, so we cannot use confluence and related properties in our proof. This is offset by the fact that we consider only typed systems.

The standard reductions of [20] are based on "outside-in" reductions. Informally, outside-in reductions are reductions in which no subterm of a term reduces before the term itself contracts unless the subterm reduces outside-in and contributes towards making the term a redex. For example, consider the PCF reduction

$$\begin{aligned}\texttt{cond}\,(\texttt{zero?}\,0)\; M\; N &\to \texttt{cond}\,\texttt{tt}\; M\; N \\ &\to M.\end{aligned}$$

The reduction is standard even though the term $\texttt{cond}\,(\texttt{zero?}\,0)\; M\; N$ contracts after its subterm $(\texttt{zero?}\,0)$, because it is the contraction of $(\texttt{zero?}\,0)$ that turns the $\texttt{cond}$ term into a redex.

There is a natural way of testing whether or not a reduction is outside-in: first, identify "outermost" subterms that contract; each of these identifies subterms that must reduce before the outer subterm itself contracts. By iterating the process, we can identify a subterm or subterms that must reduce before any others, if the reduction is to be outside-in. This idea is the basis of our definition of standard reductions.

For each term in a reduction, we identify a *principal redex*, and call a reduction standard if the redex contracted at each step is the principal redex. For the pure λ-calculus, the principal redex for some M_i will simply be the leftmost redex of M_i contracted in the reduction.

For systems with constants, we must allow reductions to take place in the critical arguments of some δ-terms. To find the principal redex, then, we start by considering the leftmost contracted subterm; if it is a δ-term, we then consider critical arguments in which contractions take place, etc. Eventually, consideration of these *preprincipal* subterms leads to the principal redex.

DEFINITION C.19. *Let M_i be a term in a reduction path $\boldsymbol{\sigma}$,*

$$\boldsymbol{\sigma} : M_1 \xrightarrow{\Delta_1} M_2 \xrightarrow{\Delta_2} M_3 \xrightarrow{\Delta_3} \cdots .$$

The subterms of M_i that are preprincipal in $\boldsymbol{\sigma}$ *are defined inductively:*

(i) *If Δ is the leftmost subterm of M_i contracted in $\boldsymbol{\sigma}$, then Δ is preprincipal in $\boldsymbol{\sigma}$.*

(ii) *If $\delta_\theta\langle\vec{A},\vec{B}\rangle$ is preprincipal and θ-contracted in $\boldsymbol{\sigma}$, and Δ is the leftmost contracted subterm of $\vec{A}$, then Δ is preprincipal in $\boldsymbol{\sigma}$.*

We write $\mathrm{pp}_\sigma(\Delta)$ *if Δ is preprincipal in $\boldsymbol{\sigma}$.*

This next lemma is essential in showing an important property of the preprincipal subterms: they are linearly ordered by $\subset$ (see the following note).

LEMMA C.20. *Let M_i be a term in a reduction path $\boldsymbol{\sigma}$,*

$$\boldsymbol{\sigma} : M_1 \xrightarrow{\Delta_1} M_2 \xrightarrow{\Delta_2} M_3 \xrightarrow{\Delta_3} \cdots ,$$

and let Δ be a preprincipal subterm of M_i. If $\Delta \not\equiv \Delta_i$, then Δ is not contained in Δ_i, and has a unique, preprincipal descendant $\Delta' \subset M_{i+1}$.

Proof. This is a proof by induction on how $\mathrm{pp}_\sigma(\Delta)$.

(i) $\mathrm{pp}_\sigma(\Delta)$ because Δ is the leftmost contracted subterm of M_i. Clearly Δ_i does not contain Δ, else Δ would not be leftmost. So either $\Delta_i \subset \Delta$, or Δ_i is disjoint and to the right of Δ. In either case, Δ has a unique descendant Δ' in M_{i+1}. Furthermore, the contraction of Δ_i can only introduce terms inside of or to the right of Δ', and so Δ' must be the leftmost contracted subterm of M_{i+1}. Thus $\mathrm{pp}_\sigma(\Delta')$.

(ii) $\mathrm{pp}_\sigma(\Delta)$ because M_i contains a preprincipal, θ-contracted subterm, $\delta_\theta\langle\vec{A},\vec{B}\rangle$, where Δ is the leftmost subterm of $\vec{A}$ contracted in $\boldsymbol{\sigma}$.
Now $\Delta_i \not\equiv \delta_\theta\langle\vec{A},\vec{B}\rangle$, or else by Note C.2(ii), Δ would have no descendant in M_{i+1}, contradicting the fact that it is contracted in $\boldsymbol{\sigma}$.
Then by induction, Δ_i does not contain $\delta_\theta\langle\vec{A},\vec{B}\rangle$. So either Δ_i is in $\vec{A}$, is in $\vec{B}$, or is entirely disjoint. If Δ_i is in $\vec{A}$, it cannot contain Δ because Δ is leftmost. And if Δ_i is in $\vec{B}$ or is disjoint, it clearly does not contain Δ.
Again by induction, $\delta_\theta\langle\vec{A},\vec{B}\rangle$ has a unique, preprincipal descendant, which must be of the form $\delta_\theta\langle\vec{A}',\vec{B}'\rangle$. And since Δ is not contained in Δ_i, Δ must have unique descendant Δ' that is the leftmost contracted subterm of $\vec{A}'$. And therefore $\mathrm{pp}_\sigma(\Delta')$. □

NOTE C.21.

(i) *By Lemma* C.20, *every preprincipal subterm contracts exactly once in $\boldsymbol{\sigma}$. Thus the θ and $\vec{A}$ of Definition* C.19(ii) *are unique.*

(ii) *By* (i), *we conclude that if* Δ_1 *and* Δ_2 *are distinct, preprincipal subterms of* M_i*, then either* $\Delta_1 \subset \Delta_2$ *or* $\Delta_2 \subset \Delta_1$.

DEFINITION C.22. *Suppose* $\boldsymbol{\sigma}$ *is a reduction path,*

$$\boldsymbol{\sigma} : M_1 \overset{\Delta_1}{\to} M_2 \overset{\Delta_2}{\to} M_3 \overset{\Delta_3}{\to} \cdots.$$

(i) *We define the* principal redex $\mathrm{pr}_{\sigma}(M_i)$ *to be the innermost preprincipal subterm of* M_i. *By Note* C.21(ii), *this is well defined.*

(ii) *We say* $\boldsymbol{\sigma}$ *is a* standard reduction *if for all* i, $\Delta_i \equiv \mathrm{pr}_{\sigma}(M_i)$.

In Definition 4.3, we gave a simple definition of "standard reductions to ground constants." The standard reductions we define here are much more general: they apply to reductions between arbitrary terms, and even to infinite reductions.

It is not immediately obvious that the general definition we give here is equivalent to our earlier definition in those cases where the earlier definition applies. This equivalence is established by the next two lemmas.

LEMMA C.23. *Suppose* $\boldsymbol{\sigma} : C[M_1] \overset{\Delta_1}{\to} C[M_2] \overset{\Delta_2}{\to} \cdots \overset{\Delta_{n-1}}{\to} C[M_n] \to \cdots$ *is a standard reduction,* $\Delta_i \subset M_i$ *for* $1 \leq i < n$, $C[\cdot]$ *is a context with a single hole, and no subterm of* M_n *contracts in* $\boldsymbol{\sigma}$. *Then the reduction* $\boldsymbol{\sigma}' : M_1 \overset{\Delta_1}{\to} M_2 \overset{\Delta_2}{\to} \cdots \overset{\Delta_{n-1}}{\to} M_n$ *is a standard reduction.*

Proof. Suppose $\Delta \subset M_i$. Since no subterm of M_n contracts in $\boldsymbol{\sigma}$, we have

(8) $$\Delta \text{ contracts in } \boldsymbol{\sigma} \text{ iff } \Delta \text{ contracts in } \boldsymbol{\sigma}'.$$

We now show that if Δ is preprincipal in $\boldsymbol{\sigma}$, then Δ is preprincipal in $\boldsymbol{\sigma}'$ by induction on the definition of preprincipal.

- Δ is the leftmost subterm of $C[M_i]$ contracted in $\boldsymbol{\sigma}$. Then Δ is the leftmost subterm of M_i contracted in $\boldsymbol{\sigma}$. And then by (8), Δ is the leftmost subterm of M_i contracted in $\boldsymbol{\sigma}'$, so Δ is preprincipal in $\boldsymbol{\sigma}'$.
- Δ is the leftmost subterm of $\vec{A}$ contracted in $\boldsymbol{\sigma}$, where $\Delta' = \delta_\theta\langle \vec{A}, \vec{B}\rangle$ is preprincipal in $\boldsymbol{\sigma}$.
  If $\Delta' \subset M_i$, then by induction, Δ' is preprincipal in $\boldsymbol{\sigma}'$. And by (8), Δ is the leftmost subterm of $\vec{A}$ contracted in $\boldsymbol{\sigma}'$, so Δ is preprincipal in $\boldsymbol{\sigma}'$.
  Otherwise, M_i is properly contained by Δ'. Then we must have $M_i \subset A_j$ for some $A_j \in \vec{A}$. So Δ is the leftmost subterm of M_i contracted in $\boldsymbol{\sigma}$. And then by (8), Δ is the leftmost subterm of M_i contracted in $\boldsymbol{\sigma}'$, so Δ is preprincipal in $\boldsymbol{\sigma}'$.

Finally, we show that if Δ is principal in $\boldsymbol{\sigma}$, then Δ is principal in $\boldsymbol{\sigma}'$. This is sufficient to show that $\boldsymbol{\sigma}'$ is standard.

By way of contradiction assume Δ is principal in $\boldsymbol{\sigma}$ but not in $\boldsymbol{\sigma}'$. We have already shown that Δ is preprincipal in $\boldsymbol{\sigma}'$, so it must be that Δ is not the *innermost* preprincipal subterm of M_i in $\boldsymbol{\sigma}'$. Then Δ must be of the form $\delta_\theta\langle \vec{A}, \vec{B}\rangle$, and there must be a leftmost subterm Δ' of $\vec{A}$ contracted in $\boldsymbol{\sigma}'$. Then by (8), Δ' is the leftmost subterm of $\vec{A}$ contracted in $\boldsymbol{\sigma}$. But then Δ' would be preprincipal in $\boldsymbol{\sigma}$, contradicting that Δ is principal in $\boldsymbol{\sigma}$. □

In the following lemma, we abbreviate "standard reduction to ground constant" by SRTGC.

LEMMA C.24. *If* $\boldsymbol{\sigma} : M \twoheadrightarrow_s c$, *then* $\boldsymbol{\sigma}$ *is an* SRTGC.

Proof. By induction on the length of $\boldsymbol{\sigma}$.

If $|\boldsymbol{\sigma}| = 0$, then $\boldsymbol{\sigma}$ must be of the form $c \twoheadrightarrow c$, which is an SRTGC.

If $|\boldsymbol{\sigma}| > 0$, consider cases on the structure of M:

- $M \equiv (\lambda x M_1) M_2 \vec{N}$. Because $\boldsymbol{\sigma}$ ends in a ground constant, we know $(\lambda x M_1) M_2$ must be contracted in $\boldsymbol{\sigma}$. Therefore it is the principal subterm of M, and $\boldsymbol{\sigma}$ must be of the form $(\lambda x M_1) M_2 \vec{N} \to (M_1[x := M_2]) \vec{N} \twoheadrightarrow_s c$. By induction the reduction $(M_1[x := M_2]) \vec{N} \twoheadrightarrow_s c$ is an SRTGC, so $\boldsymbol{\sigma}$ is an SRTGC.
- $M \equiv \delta \vec{N}$. Since $\boldsymbol{\sigma}$ ends in c, we know the head δ must be contracted. That is, $\delta \vec{N}$ must be of the form $\delta_\theta \langle \vec{C}, \vec{D} \rangle \vec{E}$, where $\Delta \equiv \delta_\theta \langle \vec{C}, \vec{D} \rangle$ is the leftmost subterm of M contracted (by θ) in $\boldsymbol{\sigma}$. By the definition of standard reductions, the only subterms that contract before Δ in $\boldsymbol{\sigma}$ are contained in $\vec{C}$, and moreover reduction in $\vec{C}$ proceeds from left to right. Therefore, $\boldsymbol{\sigma}$ is of the form

$$\begin{array}{lccl} \boldsymbol{\sigma}_1 : & \delta_\theta \langle C_1 C_2 \cdots C_n, \vec{D} \rangle \vec{E} & \twoheadrightarrow & \delta_\theta \langle c_1 C_2 \cdots C_n, \vec{D} \rangle \vec{E} \\ \boldsymbol{\sigma}_2 : & & \twoheadrightarrow & \delta_\theta \langle c_1 c_2 \cdots C_n, \vec{D} \rangle \vec{E} \\ \vdots & & \twoheadrightarrow & \cdots \\ \boldsymbol{\sigma}_n : & & \twoheadrightarrow & \delta_\theta \langle c_1 c_2 \cdots c_n, \vec{D} \rangle \vec{E} \\ & & \to_\theta & P_\theta(\vec{D}) \vec{E} \\ & & \twoheadrightarrow_s & c. \end{array}$$

By Lemma C.23, the reductions $\boldsymbol{\sigma}'_i : C_i \twoheadrightarrow c_i$ (derived from $\boldsymbol{\sigma}_i$ in the obvious way) are standard reductions. Then by induction, each $\boldsymbol{\sigma}'_i$ is an SRTGC, and therefore $\boldsymbol{\sigma}$ is an SRTGC.
- $M \equiv x \vec{N}$. No such term can reduce to a ground constant, so we do not need to consider this case. □

The following theorem is the main result of this appendix.

THEOREM C.25 (Standardization). *If $M \twoheadrightarrow N$ is a finite reduction in a PCF-like rewrite system, then there is a standard reduction from M to N.*

C.6. Path-reduction. This section gives our proof of Standardization. It is based on a proof in [24] for the pure λ-calculus, which introduced a sort of meta-reduction: a reduction relation on reduction paths. This *path-reduction* rewrites nonstandard reductions into "more standard" reductions. The following results motivate the definition of path-reduction.

LEMMA C.26. *Let $\boldsymbol{\sigma}$ be a reduction path,*

$$\boldsymbol{\sigma} : M_1 \xrightarrow{\Delta_1} M_2 \xrightarrow{\Delta_2} M_3 \xrightarrow{\Delta_3} \cdots,$$

and let $\Delta \equiv \mathrm{pr}_\sigma(M_i)$. If $\Delta_i \not\equiv \Delta$, then Δ has a unique descendant $\Delta' \subset M_{i+1}$, and $\Delta' \equiv \mathrm{pr}_\sigma(M_{i+1})$.

Proof. Lemma C.20 proves uniqueness. To show $\Delta' \equiv \mathrm{pr}_\sigma(M_{i+1})$, by the definition of pr_σ and Lemma C.20 it suffices to note the following: if $\Delta_1 \subset \Delta_2 \subset M$ have unique descendants $\Delta'_1, \Delta'_2 \subset M'$, where $M \to M'$, then $\Delta'_1 \subset \Delta'_2$. □

COROLLARY C.27. *Suppose $\boldsymbol{\sigma}$ is a reduction path,*

$$\boldsymbol{\sigma} : M_1 \xrightarrow{\Delta_1} M_2 \xrightarrow{\Delta_2} \cdots \xrightarrow{\Delta_{n-1}} M_n.$$

Then $\boldsymbol{\sigma}$ is standard iff there is no j such that Δ_j is the descendant of $\mathrm{pr}_\sigma(M_{j-1})$.

The corollary suggests a possible way to transform a nonstandard reduction into a standard reduction: successively "swap" the contraction of a principal redex with the contraction of a nonprincipal redex at the previous step. If we reach a reduction in which each principal redex contracts as soon as it becomes principal, we will have found a standard reduction.

Definition C.28. *Suppose $\boldsymbol{\sigma}$ is a nonstandard reduction, that is, there is some j such that*

$$\boldsymbol{\sigma} : \cdots \to M_{j-1} \xrightarrow{\Delta_{j-1}} M_j \xrightarrow{\Delta_j} M_{j+1} \to \cdots,$$

where Δ_j is the descendant of $\Delta'_j \equiv \mathrm{pr}_\sigma(M_{j-1})$. The subpath

$$M_{j-1} \xrightarrow{\Delta_{j-1}} M_j \xrightarrow{\Delta_j} M_{j+1}$$

is called the path-redex at step *j. Note that Δ'_j and Δ_{j-1} do not overlap, and furthermore, by Lemma C.26, Δ_j is the unique descendant of Δ'_j. Therefore, by Lemma C.8, we can find a sequence*

$$M_{j-1} \xrightarrow{\Delta'_j} M'_j \xrightarrow{\Delta'_{j-1}} \cdots \xrightarrow{\Delta'_{j-1}} M_{j+1},$$

where the Δ'_{j-1} are the descendants of Δ_{j-1}. Such a sequence is called a path-contractum. *Finally, we define* path-reduction: *$\boldsymbol{\sigma} \underset{\text{path}}{\xrightarrow{j}} \boldsymbol{\sigma}'$ if $\boldsymbol{\sigma}'$ is obtained from $\boldsymbol{\sigma}$ by replacing the path-redex at step j by a corresponding path-contractum. We will drop the index j when convenient.*

Clearly, path-reduction preserves initial and final terms, and any path-reduction normal form is a standard reduction. Moreover, the next two lemmas show that path-reduction is strongly normalizing.

Lemma C.29. *Suppose $\boldsymbol{\sigma} \underset{\text{path}}{\xrightarrow{j}} \boldsymbol{\sigma}'$, where*

$$\begin{array}{lll} \boldsymbol{\sigma} & : & M_1 \to \cdots \to M_{j-1} \xrightarrow{\Delta_{j-1}} M_j \xrightarrow{\Delta_j} M_{j+1} \to \cdots, \\ \boldsymbol{\sigma}' & : & M_1 \to \cdots \to M_{j-1} \xrightarrow{\Delta'_j} M'_j \xrightarrow{\Delta'_{j-1}} \cdots \xrightarrow{\Delta'_{j-1}} M_{j+1} \to \cdots. \end{array}$$

Then for $i \neq j$, the following hold:

(i) *If $\Delta \subset M_i$ is not contracted in $\boldsymbol{\sigma}$, then it is not contracted in $\boldsymbol{\sigma}'$.*

(ii) *If $\Delta \subset M_i$ is contracted in $\boldsymbol{\sigma}$ and $\mathrm{pp}_\sigma(\Delta)$, then Δ is contracted in $\boldsymbol{\sigma}'$.*

(iii) *If $\Delta \subset M_i$ is preprincipal in $\boldsymbol{\sigma}$, then it is preprincipal in $\boldsymbol{\sigma}'$.*

(iv) $\mathrm{pr}_\sigma(M_i) \equiv \mathrm{pr}_{\sigma'}(M_i)$.

Proof.

(i) Just note that path-reduction only permutes the order of contraction of subterms; it does not introduce new contractions.

(ii) It is clear that if Δ contracts in $\boldsymbol{\sigma}$ and does not contract in $\boldsymbol{\sigma}'$, then Δ is either Δ_{j-1} or one of its ancestors. Thus we only need consider Δ_{j-1}.
If Δ_{j-1} does not contract in $\boldsymbol{\sigma}'$, then it must be contained in Δ'_j. But Δ'_j is the principal redex of M_{j-1}, that is, the innermost preprincipal subterm of M_{j-1}. So if Δ_{j-1} is not contracted in $\boldsymbol{\sigma}'$, it is not preprincipal in $\boldsymbol{\sigma}$.

(iii) We use induction on how $\mathrm{pp}_\sigma(\Delta)$.

1. $\mathrm{pp}_\sigma(\Delta)$ because Δ is the leftmost contracted subterm of M_i. By (ii), Δ is contracted in $\boldsymbol{\sigma}'$, and by (i), it is the leftmost contracted subterm of M_i in $\boldsymbol{\sigma}'$. Therefore $\mathrm{pp}_{\sigma'}(\Delta)$.
2. $\mathrm{pp}_\sigma(\Delta)$ because $\mathrm{pp}_\sigma(\delta_\theta\langle \vec{A}, \vec{B}\rangle)$, and Δ is the leftmost contracted subterm of $\vec{A}$. By induction, $\mathrm{pp}_{\sigma'}(\delta_\theta\langle \vec{A}, \vec{B}\rangle)$, and by (i) and (ii), Δ is the leftmost subterm of $\vec{A}$ contracted in $\boldsymbol{\sigma}'$. Therefore $\mathrm{pp}_{\sigma'}(\Delta)$.

(iv) This follows from (i), (iii), and the definition of pr_σ. □

LEMMA C.30. *If $\boldsymbol{\sigma}$ is a finite reduction, then there is no infinite path-reduction starting from $\boldsymbol{\sigma}$.*

Proof. Consider a path-reduction

$$\boldsymbol{\sigma} \equiv \boldsymbol{\sigma}_1 \xrightarrow[\text{path}]{} \boldsymbol{\sigma}_2 \xrightarrow[\text{path}]{} \boldsymbol{\sigma}_3 \xrightarrow[\text{path}]{} \cdots.$$

It is not hard to see that the reduction could have been carried out in the labelled system; that is, if $\boldsymbol{\sigma}'_i$ is a labelled reduction such that $|\boldsymbol{\sigma}'_i| \equiv \boldsymbol{\sigma}_i$, and $\boldsymbol{\sigma}_i \xrightarrow[\text{path}]{j} \boldsymbol{\sigma}_{i+1}$, then there is a labelled reduction $\boldsymbol{\sigma}'_{i+1}$ such that $|\boldsymbol{\sigma}'_{i+1}| \equiv \boldsymbol{\sigma}_{i+1}$, and $\boldsymbol{\sigma}'_i \xrightarrow[\text{path}]{j} \boldsymbol{\sigma}'_{i+1}$. Thus we can find labelled reductions $\boldsymbol{\sigma}'_1, \boldsymbol{\sigma}'_2, \boldsymbol{\sigma}'_3, \ldots$ such that $|\boldsymbol{\sigma}'_i| \equiv \boldsymbol{\sigma}_i$, and

$$\boldsymbol{\sigma}'_1 \xrightarrow[\text{path}]{} \boldsymbol{\sigma}'_2 \xrightarrow[\text{path}]{} \boldsymbol{\sigma}'_3 \xrightarrow[\text{path}]{} \cdots.$$

And because labelled reduction is strongly normalizing and each $\boldsymbol{\sigma}'_i$ begins with the same $\mathcal{L}_{\mathbb{N}}$ term, each $\boldsymbol{\sigma}_i$ is finite.

Furthermore, the path-reduction can be thought of as constructing a tree of terms, with each path from root to leaf corresponding to a reduction $\boldsymbol{\sigma}_i$. Each contracted path-redex introduces a branching in the tree. For example, if $\boldsymbol{\sigma}_i \xrightarrow[\text{path}]{j} \boldsymbol{\sigma}_{i+1}$, then the root-to-leaf path corresponding to $\boldsymbol{\sigma}_{i+1}$ is obtained by branching off of the root-to-leaf path of $\boldsymbol{\sigma}_i$ at depth $j-1$. The situation is depicted in the following figure, where the root of the tree is displayed at the left and the leaves are displayed at the right:

$$\begin{tikzcd}
M_1 \arrow[r] & \cdots \arrow[r] & M_{j-1} \arrow[r, "\Delta_{j-1}"] \arrow[d, "\Delta'_j"] & M_j \arrow[r, "\Delta_j"] & M_{j+1} \arrow[r] & \cdots \arrow[r] & M_n \, : \, \sigma_i \\
 & & M'_j \arrow[r, "\Delta'_{j-1}"] & \cdots \arrow[r, "\Delta'_{j-1}"] & M_{j+1} \arrow[r] & \cdots \arrow[r] & M_n \, : \, \sigma_{i+1}.
\end{tikzcd}$$

By Lemma C.29(iv), the tree is a binary tree, and we have just seen that there is no infinite path from the root. Then by König's Lemma, the tree is finite, so the number of different reductions given by the tree must be finite. □

Proof of Theorem C.25 *(Standardization).* If $\boldsymbol{\sigma} : M \twoheadrightarrow N$ is a finite reduction in a PCF-like system, we can obtain a standard reduction from M to N just by finding a path-reduction normal form of $\boldsymbol{\sigma}$. □

The following results are included for completeness.

LEMMA C.31. *Path-reduction is confluent.*

Proof. Suppose $\boldsymbol{\sigma} \xrightarrow[\text{path}]{i} \boldsymbol{\sigma}_i$ and $\boldsymbol{\sigma} \xrightarrow[\text{path}]{j} \boldsymbol{\sigma}_j$, where $i \neq j$. We show that there is a $\boldsymbol{\sigma}'$ such that $\boldsymbol{\sigma}_i \xrightarrow[\text{path}]{} \boldsymbol{\sigma}'$, and $\boldsymbol{\sigma}_j \xrightarrow[\text{path}]{} \boldsymbol{\sigma}'$. Thus the reflexive closure of path-reduction satisfies the diamond property, and therefore path-reduction is confluent (cf. [4, Lem. 3.2.2]).

First, note that if $|i-j| \geq 2$, then the path-redexes at steps i and j are disjoint. In this case, we may contract the redexes in any order and reach a common $\boldsymbol{\sigma}'$.

Finally, we show $|i-j| \geq 2$. By way of contradiction, assume that $i = j+1$. Then $M_{j-1} \xrightarrow{\Delta_{j-1}} M_j \xrightarrow{\Delta_j} M_i \xrightarrow{\Delta_i} M_{i+1}$ is a subpath of $\boldsymbol{\sigma}$. But by the definition of path-redex, Δ_j must be the principal redex of M_j and Δ_i must be the descendent of the principal redex of M_j. This is impossible because Δ_j has no descendants in M_i (cf. Note C.2(i)). □

COROLLARY C.32. *Path-reduction normal forms are unique.*

Acknowledgments. We are grateful to G. Berry, B. Bloom, P.-L. Curien, J.-J. Lévy, G. D. Plotkin, and Scott Smith for helpful discussions. The comments of a referee motivated some technical improvements.

REFERENCES

[1] S. Abramsky, *The lazy lambda calculus*, in Research Topics in Functional Programming, D. L. Turner, ed., Addison–Wesley, Reading, MA, 1989.

[2] ———, *Domain theory in logical form*, Ann. Pure Appl. Logic, 51 (1991), pp. 1–77.

[3] S. Abramsky, P. Malacaria, and R. Jagadeesan, *Full abstraction for PCF*, in Theoretical Aspects of Computer Software, M. Hagiya and J. C. Mitchell, eds., Lecture Notes in Comput. Sci. 789, Springer-Verlag, Berlin, New York, Heidelberg, 1994, pp. 1–15.

[4] H. P. Barendregt, *The Lambda Calculus: Its Syntax and Semantics*, Studies in Logic and the Foundations of Mathematics, vol. 103, 2nd ed., North–Holland, Amsterdam, 1984.

[5] G. Berry, *Séquentialité de l'evaluation formelle des lambda-expressions*, in Program Transformations, 3ème Colloque International sur la Programmation, B. Robinet, ed., 1978, pp. 67–80.

[6] ———, *Stable models of typed lambda-calculi*, in Automata, Languages, and Programming: 5th Colloquium, G. Ausiello and C. Böhm, eds., Lecture Notes in Comput. Sci. 62, Springer-Verlag, Berlin, New York, Heidelberg, pp. 72–89.

[7] G. Berry and P.-L. Curien, *Sequential algorithms on concrete data structures*, Theoret. Comput. Sci., 20 (1982), pp. 265–321.

[8] ———, *Theory and practice of sequential algorithms: The kernel of the programming language CDS*, in Algebraic Methods in Semantics, M. Nivat and J. C. Reynolds, eds., Cambridge University Press, Cambridge, UK, 1985, pp. 35–87.

[9] G. Berry, P.-L. Curien, and J.-J. Lévy, *Full abstraction for sequential languages: The state of the art*, in Algebraic Methods in Semantics, M. Nivat and J. C. Reynolds, eds., Cambridge University Press, Cambridge, UK, 1985, pp. 89–132.

[10] B. Bloom, *Can LCF be topped? Flat lattice models of typed lambda calculus (preliminary report)*, in 3rd Annual Symposium on Logic in Computer Science [22], IEEE Press, Piscataway, NJ, 1988, pp. 282–295.

[11] ———, *Can LCF be topped?: Flat lattice models of typed λ-calculus*, Inform. and Comput., 87 (1990), pp. 263–300.

[12] B. Bloom, S. Istrail, and A. R. Meyer, *Bisimulation can't be traced*, J. Assoc. Comput. Mach., 42 (1995), pp. 232–268.

[13] A. Bucciarelli and T. Ehrhard, *Extensional embedding of a strongly stable model of PCF*, in Automata, Languages, and Programming: 18th Colloquium, Lecture Notes in Comput. Sci. 510, Springer-Verlag, Berlin, New York, Heidelberg, 1991.

[14] ———, *Sequentiality in an extensional framework*, Inform. and Comput., 110 (1994), pp. 265–296.

[15] R. Cartwright, P.-L. Curien, and M. Felleisen, *Fully abstract semantics for observably sequential languages*, Inform. and Comput., 111 (1994).

[16] P.-L. Curien, *Categorical Combinators, Sequential Algorithms and Functional Programming*, 2nd ed., Birkhäuser, Basel, Boston, 1993.

[17] J.-Y. Girard, *The system F of variable types, fifteen years later*, Theoret. Comput. Sci., 45 (1986), pp. 152–192.

[18] J. R. Hindley and J. P. Seldin, *Introduction to Combinators and λ-calculus*, London Math. Soc. Student Texts, vol. 1, Cambridge University Press, Cambridge, UK, 1986.

[19] D. J. Howe, *Equality in lazy computation systems*, in Proc. 4th Annual Symposium on Logic in Computer Science, IEEE Press, Piscataway, NJ, 1989, pp. 198–203.

[20] G. Huet and J.-J. Lévy, *Computations in nonambiguous term rewriting systems*, Tech. Report 359, INRIA, Rocquencourt, France, 1979.

[21] J. Hyland and C.-H. Ong, *Pi-calculus, dialogue games and PCF*, in Conference Record of FPCA '95: Conference on Functional Programming Languages and Computer Architecture, Association for Computing Machinery, New York, 1995, pp. 96–107.

[22] *3rd Annual Symposium on Logic in Computer Science*, IEEE Press, Piscataway, NJ, 1988.

[23] T. Jim and A. R. Meyer, *Communication in the* Types *electronic forum*, June 17, 1989.

[24] J. W. Klop, *Combinatory reduction systems*, Tract 127, Mathematisch Centrum, Amsterdam, 1980.

[25] I. Mason and C. Talcott, *Programming, transforming, and proving with function abstractions and memories*, in Automata, Languages, and Programming: 16th International Col-

loquium, G. Ausiello, M. Dezani-Ciancaglini, and S. R. D. Rocca, eds., Lecture Notes in Comput. Sci. 372, Springer-Verlag, Berlin, New York, Heidelberg, 1989.

[26] A. R. Meyer, *What is a model of the lambda calculus?*, Inform. and Control, 52 (1982), pp. 87–122.

[27] ———, *Semantical paradigms: Notes for an invited lecture, with two appendices by Stavros Cosmadakis*, in 3rd Annual Symposium on Logic in Computer Science [22], IEEE Press, Piscataway, NJ, pp. 236–253.

[28] R. Milner, *Fully abstract models of the typed lambda calculus*, Theoret. Comput. Sci., 4 (1977), pp. 1–22.

[29] K. Mulmuley, *Full Abstraction and Semantic Equivalence*, ACM Doctoral Dissertation Award, 1986, MIT Press, 1987.

[30] C.-H. L. Ong, *The lazy lambda calculus: An investigation into the foundations of functional programming*, Ph.D. thesis, Imperial College, University of London, London, 1988.

[31] G. D. Plotkin, *LCF considered as a programming language*, Theoret. Comput. Sci., 5 (1977), pp. 223–256.

[32] V. Sazonov, *Expressibility of functions in D. Scott's LCF language*, Algebra i Logika, 15 (1976), pp. 308–330 (in Russian).

[33] D. S. Scott, *Continuous lattices*, in Toposes, Algebraic Geometry, and Logic, F. W. Lawvere, ed., Lecture Notes in Math. 274, Springer-Verlag, Berlin, New York, Heidelberg, 1972, pp. 97–136.

[34] ———, *Data types as lattices*, SIAM J. Comput., 5 (1976), pp. 522–587.

[35] ———, *A type theoretical alternative to CUCH, ISWIM, OWHY*, Theoret. Comput. Sci., 121 (1993), pp. 411–440.

[36] S. Smith, *From operational to denotational semantics*, in Mathematical Foundations of Programming Semantics, Lecture Notes in Comput. Sci., Springer-Verlag, Berlin, New York, Heidelberg, 1992, pp. 54–76.

[37] A. Stoughton, *Fully Abstract Models of Progamming Languages*, Research Notes in Theoretical Computer Science, Pitman/Wiley, Boston, New York, 1988; revision of Ph.D. thesis, Report CST-40-86, Department of Computer Science, University of Edinburgh, Edinburgh, 1986.

[38] C. Talcott, *Programming and proving with function and control abstractions*, Tech. Report STAN-CS-89-1288, Stanford University, Stanford, CA, 1988.

SIAM J. COMPUT.
Vol. 25, No. 4, pp. 697–708, August 1996

COMPUTING SOLUTIONS UNIQUELY COLLAPSES THE POLYNOMIAL HIERARCHY*

LANE A. HEMASPAANDRA†, ASHISH V. NAIK‡, MITSUNORI OGIHARA§, AND ALAN L. SELMAN¶

Abstract. Is there an NP function that, when given a satisfiable formula as input, outputs one satisfying assignment uniquely? That is, can a nondeterministic function cull just one satisfying assignment from a possibly exponentially large collection of assignments? We show that if there is such a nondeterministic function, then the polynomial hierarchy collapses to ZPP^{NP} (and thus, in particular, to NP^{NP}). Because the existence of such a function is known to be equivalent to the statement "every NP function has an NP refinement with unique outputs," our result provides the strongest evidence yet that NP functions cannot be refined.

We prove our result via a result of independent interest. We say that a set A is NPSV-selective (NPMV-selective) if there is a 2-ary partial NP function with unique values (a 2-ary partial NP function) that decides which of its inputs (if any) is "more likely" to belong to A; this is a nondeterministic analog of the recursion-theoretic notion of the semirecursive sets and the extant complexity-theoretic notion of P-selectivity. Our hierarchy-collapse result follows by combining the easy observation that every set in NP is NPMV-selective with the following result: If $A \in$ NP is NPSV-selective, then $A \in (\mathrm{NP} \cap \mathrm{coNP})/\mathrm{poly}$. Relatedly, we prove that if $A \in$ NP is NPSV-selective, then A is Low_2.

We prove that the polynomial hierarchy collapses even further, namely to NP, if all coNP sets are NPMV-selective. This follows from a more general result we prove: Every self-reducible NPMV-selective set is in NP.

Key words. computational complexity, semidecision algorithms, nonuniform complexity, lowness

AMS subject classifications. 68Q15, 68Q10, 03D10, 03D15

1. Introduction.

Valiant and Vazirani's [42] result that, in their words, "NP is as easy as detecting unique solutions," has rightly been the focus of great attention. Their breakthrough—a proof that every NP set *probabilistically* reduces to "detecting unique solutions" (technically, reduces to every solution to the promise problem (1SAT, SAT))—is one of the dual pillars on which Toda's [40] PH $\subseteq P^{PP}$ paper rests, as do later papers extending Toda's result [41] and studying the complexity of function inversion [43, 1].

Selman ([38]; see also [12]) raised a related question that may be equally compelling, since he showed that a resolution would provide insight into the invertibility of honest polynomial-time functions and into the relationship between single-valued and multivalued functions. He asked whether the following hypothesis is true.

HYPOTHESIS 1.1. *There is a* single-valued NP *function f such that for each formula $F \in$ SAT, $f(F)$ is a satisfying assignment of F.*

Clearly, Hypothesis 1.1 is true if NP = coNP. However, since both Fenner et al. [12] and Selman [38] suspected that Hypothesis 1.1 fails, perhaps a more interesting issue is that of the evidential weight in that direction. In fact, little is currently known to indicate that Hypothesis 1.1 fails. The totality of current evidence seems to be the fact that Hypothesis 1.1 fails relative to a random oracle [33] and the result of Selman [38] that if Hypothesis 1.1 holds,

*Received by the editors May 24, 1994; accepted for publication (in revised form) November 18, 1994.

†Department of Computer Science, University of Rochester, Rochester, NY 14627. The research of this author was supported in part by NSF grants CCR-8957604, INT-9116781/JSPS-ENG-207, and CCR-9322513. This research was done in part while this author was visiting the University of Electro-Communications, Tokyo, Japan, and the Tokyo Institute of Technology, Tokyo, Japan.

‡Department of Computer Science, State University of New York at Buffalo, Buffalo, NY 14260. Current address: Department of Computer Science, University of Chicago, Chicago, IL 60637. The research of this author was supported in part by NSF grant CCR-9002292.

§Department of Computer Science, University of Rochester, Rochester, NY 14627. The research of this author was supported in part by NSF grants CCR-9002292 and INT-9116781/JSPS-ENG-207. This work was done in part while visiting SUNY–Buffalo and while this author was at the University of Electro-Communications, Tokyo, Japan.

¶Department of Computer Science, State University of New York at Buffalo, Buffalo, NY 14260. The research of this author was supported in part by NSF grants CCR-9002292, INT-9123551, and CCR-9400229.

then there are two disjoint NP-Turing-complete sets such that every set that separates them is NP-hard.

Since Hypothesis 1.1 is implied by NP = coNP, one might hope that Hypothesis 1.1 also implies a collapse of the polynomial hierarchy. The main result of this paper provides strong evidence that Hypothesis 1.1 fails: Hypothesis 1.1 implies that the polynomial hierarchy collapses to $\mathrm{ZPP}^{\mathrm{NP}}$ (and thus, in particular, to its second level, $\mathrm{NP}^{\mathrm{NP}}$). Equivalently, if all honest polynomial-time-computable functions are NPSV-invertible, then the polynomial hierarchy collapses to $\mathrm{ZPP}^{\mathrm{NP}}$.

We obtain our result from a surprising and seemingly little-related direction: selectivity. Selectivity is a notion of generalized membership testing; selective sets have functions that choose which of any two input elements is the "more likely" to be in the set. Sets selective with respect to recursive selector functions were introduced by Jockush [20] and are called the *semirecursive* sets. Sets selective with respect to deterministic polynomial-time selector functions were introduced by Selman [36] and are called the P-selective sets; sets selective with respect to single-valued total NP functions were introduced and studied by Hemaspaandra et al. [19]. Recently, there has been a surge of interest in selective sets, and advances have catalyzed further advances (see the survey [9]).

In this paper, we extend the notion of selectivity, in the natural way, to functions that may be partial and/or multivalued. Important function classes of these sorts are the single-valued partial NP functions (NPSV), the multivalued partial NP functions (NPMV), and the multivalued total NP functions (NPMV_t). Although it is easily observed that all NP sets are NPMV-selective, we will prove the following result.

($\star\star$) If all NP sets are NPSV-selective, then the polynomial hierarchy collapses to $\mathrm{ZPP}^{\mathrm{NP}}$.

It follows easily that Hypothesis 1.1 implies this same collapse.

Result ($\star\star$) is proven via the following result, which is of interest in its own right.

(1) The NPSV-selective sets in NP are in $(\mathrm{NP} \bigcap \mathrm{coNP})/\mathrm{poly}$.

$(\mathrm{NP} \bigcap \mathrm{coNP})/\mathrm{poly}$ is the class of sets (see [14]) accepted, aided by a small amount of "advice," by machines that robustly behave as $\mathrm{NP} \bigcap \mathrm{coNP}$ machines. We also prove the following related result.

(2) The NPSV-selective sets in NP are Low_2.

That is, for each such set A, $\mathrm{NP}^{\mathrm{NP}^A} = \mathrm{NP}^{\mathrm{NP}}$. Though NPSV functions lack totality, the proofs of (1) and (2) show that we can nonetheless get the *effect* of totality in the cases that count—in particular, the definition of selectivity forces the functions to be defined whenever at least one input is in the fixed selective set. This will allow us to establish that the NPSV-selective sets in NP have lowness and advice class results just as strong as those shown by [19] for the NPSV_t-selective sets in NP. This advance is important because results about NPSV_t-selective sets offer no help in discrediting Hypothesis 1.1 but results about NPSV-selective sets do.

For coNP (and thus all higher levels of the polynomial hierarchy), an even stronger consequence can be obtained: All coNP sets are NPMV-selective if and only if NP = coNP. This result itself is a corollary of a more general result we prove: Every self-reducible NPMV-selective set is in NP. This contrasts with Buhrman, van Helden, and Torenvliet's result [8] that self-reducible P-selective sets are in P and with the result announced in [18] that self-reducible NPMV_t-selective sets are in $\mathrm{NP} \cap \mathrm{coNP}$.

2. Definitions. Our alphabet will be $\Sigma = \{0, 1\}$. Let our pairing function $\langle \cdot \cdot \cdot \rangle$ be any "multi-arity onto," polynomial-time computable, polynomial-time invertible function (that is, the ranges of different arities are disjoint, and the union over all arities covers Σ^*; see, e.g., [16]).

For each partial, multivalued function f, *set-*$f(x)$ denotes the set of values of f on input x. If $f(x)$ is undefined, then *set-*$f(x) = \emptyset$. We will also use this notation for partial single-valued functions to avoid ambiguity regarding equality tests between potentially undefined values. For any two partial, multivalued functions f and g, we say that f is a *refinement* of g if, for all x, it holds that

1. $f(x)$ is defined if and only if $g(x)$ is defined, and
2. *set-*$f(x) \subseteq$ *set-*$g(x)$.

We extend notions of selectivity [36, 19] to multivalued and/or partial functions.

DEFINITION 2.1. *Let $\mathcal{FC}$ be any class of functions (possibly multivalued and/or partial). A set A is $\mathcal{FC}$-selective if there is a function $f \in \mathcal{FC}$ such that for every x and y, it holds that*

$$\textit{set-}f(x, y) \subseteq \{x, y\}, \textit{ and}$$

$$\textit{if } \{x, y\} \cap A \neq \emptyset, \textit{ then set-}f(x, y) \neq \emptyset \textit{ and set-}f(x, y) \subseteq A.$$

By $\mathcal{FC}$-sel *we denote the class of sets that are $\mathcal{FC}$-selective.*

We will be interested, in particular, in the following classes of functions.

DEFINITION 2.2 (see [6]).

1. NPMV *is the class of partial, multivalued functions f for which there is a nondeterministic polynomial-time machine N such that for every x, it holds that*
   (a) *$f(x)$ is defined if and only if $N(x)$ has at least one accepting computation path, and*
   (b) *for every y, $y \in$ set-$f(x)$ if and only if there is an accepting computation path of $N(x)$ that outputs y.*
2. NPMV_t *is the class of total, multivalued functions in* NPMV.
3. NPSV *is the class of partial, single-valued functions in* NPMV.
4. NPSV_t *is the class of total, single-valued functions in* NPMV.

Hypothesis 1.1 says that there is a partial function f in NPSV such that for every formula F in SAT, $f(F)$ is a satisfying assignment for F. It is trivial to observe that there is an NPMV function that finds *all* satisfying assignments of an input formula. Thus the true complexity issue here is not that of the complexity of finding satisfying assignments but is rather that of the complexity of *thinning down to one* the satisfying assignment set. Hypothesis 1.1 is equivalent to the assertion that all NPMV functions have refinements in NPSV (see [38]; see also Proposition 3.1 in §3). We observe (Proposition 3.1) that Hypothesis 1.1 holds if and only if SAT is NPSV-selective.

Karp and Lipton introduced the following notion of being computable in a class supplemented by a small amount of extra information.

DEFINITION 2.3 (see [21]). *For any class of sets $\mathcal{C}$, $\mathcal{C}$/*poly *denotes the class of sets L for which there exist a set $A \in \mathcal{C}$ and a polynomially length-bounded function $h : \Sigma^* \to \Sigma^*$ such that for every x, it holds that*

$$x \in L \textit{ if and only if } \langle x, h(0^{|x|}) \rangle \in A.$$

We will be particularly interested in the advice classes NP/poly, coNP/poly, and (NP $\bigcap$ coNP)/poly. It is not known whether NP/poly $\bigcap$ coNP/poly = (NP $\bigcap$ coNP)/poly, though Fenner et al. [11] have constructed an oracle relative to which the classes differ (see also the structural results of [14]).

Next we define lowness and extended lowness.

DEFINITION 2.4.

1. [34] *For each $k \geq 1$, define* $\mathrm{Low}_k = \{L \in \mathrm{NP} \mid \Sigma_k^{p,L} = \Sigma_k^p\}$, *where the Σ_k^p are the Σ levels of the polynomial hierarchy* [30, 39].
2. [27, 4] *For each $k \geq 2$, define* $\mathrm{ExtendedLow}_k = \{L \mid \Sigma_k^{p,L} = \Sigma_{k-1}^{p,\mathrm{SAT}\oplus L}\}$. *For each $k \geq 3$, define* $\mathrm{ExtendedLow}\Theta_k = \{L \mid \mathrm{P}^{(\Sigma_{k-1}^{p,L})[\mathcal{O}(\log n)]} \subseteq \mathrm{P}^{(\Sigma_{k-2}^{p,\mathrm{SAT}\oplus L})[\mathcal{O}(\log n)]}\}$. *The $[\mathcal{O}(\log n)]$ indicates that at most $\mathcal{O}(\log n)$ queries are made to the oracle.*

Hemaspaandra et al. [19] noted the following lowness and nonuniform class results for NPSV_t-sel: $\mathrm{NPSV}_t\text{-sel} \subseteq (\mathrm{NP} \bigcap \mathrm{coNP})/\mathrm{poly}$, $\mathrm{NPSV}_t\text{-sel} \bigcap \mathrm{NP} \subseteq \mathrm{Low}_2$, and $\mathrm{NPSV}_t\text{-sel} \subseteq \mathrm{ExtendedLow}\Theta_3$.

Finally, we define "promise problems" [10] corresponding to selectivity. Informally, a solution to the promise problem PP-A [37, 28] will—if the promise is met such that exactly one of x and y is in A—contain $\langle x, y\rangle$ exactly if $x \in A$.

DEFINITION 2.5 (see [37]; see also [28]). *Given any set A, we say that a set B* is a solution *to* PP-A *if for every* $\langle x, y\rangle$ *such that exactly one of x and y is in A,* $\langle x, y\rangle \in B$ *if and only if* $x \in A$.

3. Unique solutions collapse the polynomial hierarchy. We first note a connection between refinements of NPMV functions, NPSV-selectivity, and inversion of polynomial-time-functions. As is standard, we say a total polynomial-time-computable function f is *honest* if there is a polynomial q such that, for all x, $q(|f(x)|) \geq |x|$. If f is a (possibly non-one-to-one, possibly non-onto) total polynomial-time computable function, we say that f is $\mathcal{C}$-invertible if there is a single-valued function g in $\mathcal{C}$ such that $(\forall x)\,[(x \notin \mathrm{range}(f) \Rightarrow g(x) = \mathtt{undef})$ and $(x \in \mathrm{range}(f) \Rightarrow f(g(x)) = x)]$ (see [2, 15, 43, 38] for a detailed discussion of invertibility). Observe that f is $\mathcal{C}$-invertible if and only if the partial multivalued function f^{-1} has a single-valued refinement in $\mathcal{C}$. NP2V is the class of all NPMV functions f such that $(\forall x)\,[\,||\,\textit{set-}f(x)\,||\, \leq 2]$.

PROPOSITION 3.1 (see also [38]). *The following are equivalent:*

1. *Hypothesis* 1.1 *holds.*
2. *All* NPMV *functions have* NPSV *refinements.*
3. *All* NP2V *functions have* NPSV *refinements.*
4. SAT *is* NPSV*-selective.*
5. *All* NP *sets are* NPSV*-selective.*
6. *All honest* FP *functions are* NPSV*-invertible.*

Proof. The reader may easily observe that every set in NP is NP2V-selective. Note also that any NPSV refinement of an NPMV-selector for a set is itself an NPSV-selector for the set. Thus part 3 implies part 5. Clearly, part 5 implies part 4, and part 2 implies part 3. Part 4 implies part 1 since an NPSV function f' that is an NPSV-selector for SAT could be used to create the function f from the statement of Hypothesis 1.1 as follows. Let f be the function that (a) on an input formula F simulates f' applied to the top node of F's 2-disjunctive-self-reduction tree—and then each path of (the simulation of) f that gets an output simulates f applied to 2-disjunctive-self-reduction of the output node, and so on—and (b) at any thusly reached leaf of the self-reduction tree checks that the leaf is a satisfying assignment and outputs it if it is. Finally, Selman [38] has noted that parts 1, 2, and 6 are equivalent; part 6 is equivalent by combining [38, Exercise 5] with the comment in the last paragraph of [38, §1.2]. □

Naik, Regan, Royer, and Selman (in a work in preparation) have noted that this behavior applies not just to the classes mentioned here but to any class having certain nice closure properties.

Our hierarchy result will follow easily from our study of the lowness and circuit properties of the new selectivity classes we've mentioned—the NPSV-selective sets, the NPMV-selective sets, and the NPMV_t-selective sets. We now turn to this study, emphasizing the NPSV-selective sets. Clearly, NPMV-sel (and thus NPSV-sel) is contained in NP/poly and NPMV_t-sel is contained in $\mathrm{NP/poly} \bigcap \mathrm{coNP/poly}$ via the use of a standard divide-and-conquer approach to find an appropriate advice set, similar to the approach in Ko's proof [22] that the P-selective sets are in P/poly (see also the proofs of Theorems 3.2 and 3.7). We conclude, via the extended lowness of $\mathrm{NP/poly} \bigcap \mathrm{coNP/poly}$ (Theorem 3.4) and the lowness of $\mathrm{NP/poly} \bigcap \mathrm{coNP/poly} \bigcap \mathrm{NP} = \mathrm{coNP/poly} \bigcap \mathrm{NP}$ [44], that NPMV_t-sel is $\mathrm{ExtendedLow}_3$ and that $\mathrm{NPMV}_t\text{-sel} \bigcap \mathrm{NP}$ is Low_3. We now turn towards our main result.

THEOREM 3.2. NPSV-sel $\bigcap$ NP $\subseteq$ (NP $\bigcap$ coNP)/poly.

In the introduction, we mentioned that the key result of our proofs is the achievement, even with the partial functions, of the *effect* of totality. In the proof of Theorem 3.2, it is easy to put one's finger on the exact part of the construction that achieves this—our decision to require the advice string to encode certificates. This decision allows what would otherwise be an NP/poly $\bigcap$ coNP/poly containment to become an (NP $\bigcap$ coNP)/poly containment, since the fact that the advice contains certificates allows an NP $\bigcap$ coNP machine to verify whether or not the strings purported to be from the set in fact are from the set, and this itself allows the machine to be robustly NP $\bigcap$ coNP-like, that is, NP $\bigcap$ coNP-like for all possible advice strings, even incorrect ones.

Proof of Theorem 3.2. Let $A \in$ NP be NPSV-selective, with selector function $f \in$ NPSV. Without loss of generality, we assume f satisfies $(\forall x, y)[\textit{set-}f(x, y) = \textit{set-}f(y, x)]$, since if it doesn't, we can replace it with $f\textit{-new}(a, b) = f(\min(a, b), \max(a, b))$. It is trivial to create an appropriate advice string at lengths n for which $||A^{\leq n}|| = 0$, so we assume this is done tacitly at such lengths and consider below only the $||A^{\leq n}|| \neq 0$ case. Recall that $\textit{set-}f(x) = \{y \mid y \text{ is a value of } f(x)\}$. Let p be a monotone nondecreasing polynomial and B be a set in P witnessing that $A \in$ NP so that for every x, $x \in A$ if and only if for some $y \in \Sigma^{p(|x|)}$, $\langle x, y\rangle \in B$. Let w be a string of the form $\langle 0^n, S, T\rangle$, where S and T encode finite sets. We call w an *advice string* for n if (i) $||T|| \leq ||S|| \leq n + 1$, (ii) $S \subseteq \Sigma^{\leq n}$, (iii) $T \subseteq \Sigma^{\leq p(n)}$, and (iv) for every $y \in S$, there is some $z \in T$ such that $\langle y, z\rangle \in B$, that is, $y \in A$ is certified by z. Moreover, w is called a *good advice string* for n if it holds that

$$(*) \qquad (\forall x \in \Sigma^{\leq n})[x \in A \Rightarrow (\exists y \in S)[\textit{set-}f(x, y) = \{x\}]].$$

For every n, a good advice string for n exists. As in the case of Ko's proof that the P-selective sets are in P/poly [22], we may repeatedly choose to add to S some element of $A^{\leq n}$ that loses to at least half the elements that both are not yet in S and do not yet beat some element in S, where "x loses to y" means that $\textit{set-}f(x, y) = \{y\}$. Since $||\Sigma^{\leq n}|| < 2^{n+1}$, S will have at most $n + 1$ elements. After constructing S, for each $y \in S$, we pick up one certificate and construct T.

Clearly, the set of all advice strings is in P. Moreover, the set of all good advice strings is in coNP. As $w = \langle 0^n, S, T\rangle$ being an advice string guarantees that $S \subseteq A$, $\textit{set-}f(x, y)$ is defined for any $x \in \Sigma^{\leq n}$ and $y \in S$. So $w = \langle 0^n, S, T\rangle$ is a good advice string for n if and only if

$$w \text{ is an advice string and } (\forall x \in \Sigma^{\leq n})[x \notin A \vee (\exists y \in S)[y = x \ \vee \ y \notin \textit{set-}f(x, y)]].$$

Clearly, this is a coNP-predicate since, in particular, testing $y \notin \textit{set-}f(x, y)$ can be done via one universal quantification. However, note that if w is an advice string for n, then for every $x \in \overline{A}^{\leq n}$ and $y \in S$, $\textit{set-}f(x, y) = \{y\} \neq \{x\}$. So if $w = \langle 0^n, S, T\rangle$ is a good advice string for n, then

$$(\star) \qquad (\forall x \in \Sigma^{\leq n})[x \in A \iff (\exists y \in S)[\textit{set-}f(x, y) = \{x\}]].$$

Now define

$$A' = \{\langle x, \langle 0^{|x|}, S, T\rangle\rangle \mid \langle 0^{|x|}, S, T\rangle \text{ is an advice string for } |x| \text{ and } (\exists y \in S)[\textit{set-}f(x, y) = \{x\}]\}.$$

Note that $A' \in$ NP $\bigcap$ coNP. The containment in NP is immediate. The containment in coNP follows from the fact that, as long as $\langle 0^{|x|}, S, T\rangle$ is an advice string for $|x|$, $S \subseteq A$ guarantees that $\textit{set-}f(x, y)$ is either $\{x\}$ or $\{y\}$ for any $y \in S$.

Now for each n, define $h(0^n)$ to be the smallest good advice string for n in lexicographic order. Then by ($\star$), for every x, $x \in A$ if and only if $\langle x, h(0^{|x|})\rangle \in A'$. This proves that $A \in (\mathrm{NP} \bigcap \mathrm{coNP})/\mathrm{poly}$. □

Theorem 3.3 follows from essentially the same proof as that of Theorem 3.2.

THEOREM 3.3. NPSV-sel $\subseteq$ NP/poly $\bigcap$ coNP/poly.

This result reflects a more general behavior. We denote $\{\langle x, y\rangle \mid y \in \mathit{set}\text{-}f(x)\}$ by *graph*(f). Let $\mathcal{FC}$ be any function class (possibly partial, possibly multivalued). Let $\mathcal{C}$ be any class that has the property that for each f in $\mathcal{FC}$ it holds that *graph*$(f) \in \mathcal{C}$. Then

$$\mathcal{FC}\text{-sel} \subseteq (R^p_{dtt}(\mathcal{C}))/\text{poly}.$$

In particular, the polynomial advice represents advice strings found by divide and conquer, and the disjunctive queries determine, via the graph of the selector function, the action of the selector function on the input paired with each string in the advice set, and, additionally, the disjunctive reducer checks whether the input is one of the advice strings. The reducer accepts exactly when the input is either one of the advice strings or an output of the selector function when that function is run on the input paired with one of the advice strings (see the proofs of Theorems 3.2 above and 3.7 below). Theorem 3.3 is a specific case of this more general claim. The polynomial time-bound on the disjunctive reduction in the general claim can be replaced by a logspace bound if the pairing function used (in the definition of advice classes) is logspace invertible.

Köbler [23] has shown that (NP $\bigcap$ coNP)/poly is ExtendedLowΘ_3. An interesting question left open by Köbler's paper is whether (NP/poly) $\bigcap$ (coNP/poly) is extended low. We resolve this issue by showing that it is. It is an interesting open issue whether our result can be strengthened via the techniques of Gavaldà and Köbler [13, 23] to an ExtendedLowΘ_3 result; we conjecture that it can. In any case, in terms of the standard levels of extended lowness—ExtendedLow$_1$, ExtendedLow$_2$, ExtendedLow$_3$, ...—our ExtendedLow$_3$ result *is* optimal, since Allender and Hemaspaandra [3] have noted that even P/poly is not in ExtendedLow$_2$. We defer the proof of Theorem 3.4 to the end of this section.

THEOREM 3.4. (NP/poly) $\bigcap$ (coNP/poly) *is* ExtendedLow$_3$.

From Theorems 3.3 and 3.4, we immediately obtain the following corollary.

COROLLARY 3.5. *The* NPSV-*selective sets are* ExtendedLow$_3$.

What can be said about the lowness of the NPSV-selective sets in NP? Theorem 3.3 and Köbler's "(NP $\bigcap$ coNP)/poly $\bigcap$ NP is LowΘ_3" result imply a LowΘ_3 result. However, as the next corollary states, the NPSV-selective sets in NP are, in fact, Low$_2$. Informally, the reason for this improvement is that NPSV-selective sets have selector functions that, while perhaps partial, are sharply constrained. In particular, these functions are only partially partial. They are forced to be total whenever either of the inputs is in the given set, and, as we did also in the proof of Theorem 3.2, we exploit this conditional totality in our Low$_2$ proof below.

LEMMA 3.6 (see [28]). *If A is in Σ_i^p and B is a solution of* PP-A, *then* $\Sigma_{i+1}^{p,A} \subseteq \Sigma_{i+1}^{p,B}$.

THEOREM 3.7. *If* $A \in$ NPSV-sel $\bigcap$ NP, *then* PP-A *has a solution L that is* Low$_2$.

Corollary 3.8 follows immediately from Theorem 3.7 via Lemma 3.6.

COROLLARY 3.8. NPSV-sel $\bigcap$ NP $\subseteq$ Low$_2$.

Proof of Theorem 3.7. Let $A \in$ NPSV-sel $\cap$ NP, with selector function $f \in$ NPSV. As in the proof of Theorem 3.2, define the notion of advice strings and good advice strings. Define

$$\widehat{A} = \{\langle x, y\rangle \mid \mathit{set}\text{-}f(x, y) = \{x\} \text{ and } x \in A\}.$$

Clearly, $\widehat{A}$ is a solution of PP-A and is in NP. It suffices to show that $\Sigma_2^{p,\widehat{A}} \subseteq \Sigma_2^p$. Let $w = \langle 0^n, S, T\rangle$ be a good advice string for n. Then for every $x \in \Sigma^{\le n}$,

$$x \in A \iff (\exists y \in S)[\mathit{set}\text{-}f(x, y) = \{x\}].$$

So for every $x, y \in \Sigma^{\le n}$,

$$\begin{aligned}\langle x, y\rangle \notin \widehat{A} &\iff x \notin A \vee \textit{set-}f(x, y) \neq \{x\}\\ &\iff x \notin A \vee (x \in A \wedge \textit{set-}f(x, y) \neq \{x\})\\ &\iff (\forall z \in S)[\textit{set-}f(x, z) \neq \{x\}] \vee (x \in A \wedge \textit{set-}f(x, y) \neq \{x\})\\ &\iff (\forall z \in S)[\textit{set-}f(x, z) = \{z\}] \vee (x \in A \wedge \textit{set-}f(x, y) = \{y\}).\end{aligned}$$

Define $T = \{\langle x, y, \langle 0^n, S, T\rangle\rangle \mid |x|, |y| \leq n, w = \langle 0^n, S, T\rangle$ is an advice string for n, and either $(\forall z \in S)[\textit{set-}f(x, z) = \{z\}]$ or $x \in A \wedge \textit{set-}f(x, y) = \{y\}\}$. Then $T \in$ NP, and for every good advice string $w = \langle 0^n, S, T\rangle$ and x, y of length at most n, $\langle x, y\rangle \notin \widehat{A}$ if and only if $\langle x, y, w\rangle \in T$.

Now let $C \in \Sigma_2^{p,\widehat{A}}$ and let N_1 and N_2 be NP machines such that $C = L(N_1, L(N_2, \widehat{A}))$. There is a polynomial q such that for every x and every possible query y of N_1 on x, if N_2 on y queries $\langle u, v\rangle$, then $|u|, |v| \leq q(|x|)$. Define D to be the set of all $\langle y, \langle 0^m, S, T\rangle\rangle$ such that

- $w = \langle 0^m, S, T\rangle$ is an advice string for m and
- there is an accepting computation path π of N_2 on y such that for every query $\langle u, v\rangle$ along path π,
  - $|u|, |v| \leq m$,
  - if the answer to the query is affirmative, then $\langle u, v\rangle \in \widehat{A}$, and
  - if the answer to the query is negative, then $\langle u, v, w\rangle \in T$.

Since both $\widehat{A}$ and T are in NP, $D \in$ NP. Furthermore, if y is a query of N_1 on x, then for every good advice string w for $q(|x|)$, $N_2^{\widehat{A}}$ on y accepts if and only if $\langle y, w\rangle \in D$.

Now define E to be the set of all $\langle x, w\rangle$ such that w is an advice string for $q(|x|)$ and N_1 on x accepts if its query y is answered affirmatively if and only if $y \in D$. Since D is in NP, $E \in \Sigma_2^p$. Furthermore, for every x and good advice string w for $q(|x|)$, $\langle x, w\rangle \in E$ if and only if $x \in C$. Therefore, for every x, $x \in C$ if and only if there is a good advice string w for $q(|x|)$ such that $\langle x, w\rangle \in E$. As described in the proof of Theorem 3.2, the set of all good advice strings is in coNP. Thus $C \in \Sigma_2^p$. This proves the theorem. □

Note that every NP set is NPMV-selective. Is this also true for NPSV-selectivity? We have the following result.

THEOREM 3.9. *If* NP $\subseteq$ NPSV-sel, *then* $\text{ZPP}^{\text{NP}} = \text{PH}$.

Proof. This is a corollary of Theorem 3.2, since, extending Karp and Lipton [21], Köbler and Watanabe have proven that if NP $\subseteq$ (NP $\bigcap$ coNP)/poly $\Rightarrow$ $\text{ZPP}^{\text{NP}} = \text{PH}$ [24]. □

Note that we could immediately conclude from Corollary 3.8 the slightly weaker result that if NP $\subseteq$ NPSV-sel, then $\text{NP}^{\text{NP}} = \text{PH}$.

From Proposition 3.1 and Theorem 3.9, we have our main result, and a related result.

COROLLARY 3.10. *If Hypothesis* 1.1 *is true then* $\text{ZPP}^{\text{NP}} = \text{PH}$.

COROLLARY 3.11. *If all honest* FP *functions are* NPSV*-invertible then* $\text{ZPP}^{\text{NP}} = \text{PH}$.

Hypothesis 1.1 seems somewhat akin to the statement UP = NP in the sense that both speak of reducing a multiplicity (respectively, of values and of certificates) to a unity. However, NP might be equal to UP because of the existence of some strange machine that accepts SAT uniquely and has nothing to do with finding satisfying assignments; on the other hand, there may exist a machine that outputs satisfying assignments uniquely but "ambiguously"—along more than one computation path. Indeed, it remains an open question whether either of UP = NP and Hypothesis 1.1 implies the other. It also remains an open question whether Corollary 3.10 remains true if the hypothesis is changed to UP = NP; indeed, it is not even known whether UP = NP implies that the polynomial hierarchy collapses at any level. It is easily seen, as noted by Buhrman, Kadin, and Thierauf [7], that SAT has an NPSV refinement if and only if it has (in a certain model for oracle access to partial functions) an $\text{FP}^{\text{NPSV[1]}}$ refinement, and thus Corollary 3.10 speaks to that case.

We conclude this section with the deferred proof of Theorem 3.4.

Proof of Theorem 3.4 Let $H \in (\mathrm{NP/poly}) \bigcap (\mathrm{coNP/poly})$. Let $B \in \mathrm{NP}^{\mathrm{NP}^{\mathrm{NP}^{H}}}$ (let's say, for convenience, $B = L(N_1^{L(N_2^{L(N_3^H)})})$). We will show that $B \in \mathrm{NP}^{\mathrm{NP}^{\mathrm{SAT} \oplus H}}$.

Let S_1 (S_2) be an NP (coNP) set certifying $H \in \mathrm{NP/poly}$ ($H \in \mathrm{coNP/poly}$). Let $p(\cdot)$ be a polynomial bounding the size of the correct advice sequences for each. Let $q(\cdot)$ be a polynomial composing the polynomial running times of N_1, N_2, and N_3.

Recall that our pairing function $\langle \cdots \rangle$ is some nice, "multi-arity onto" pairing function. On input x, our base NP machine of our $\mathrm{NP}^{\mathrm{NP}^{\mathrm{SAT} \oplus H}}$ machine nondeterministically guesses strings $r_1, \ldots, r_{q(|x|)}$, and $s_1, \ldots, s_{q(|x|)}$, satisfying, for each i, $|r_i| \leq p(i)$ and $|s_i| \leq p(i)$. Via a single call to $\mathrm{NP}^{\mathrm{SAT} \oplus H}$, the base machine checks whether $r_1, \ldots, r_{q(|x|)}$ is a good advice set for helping S_1. In particular, we make one query, $\langle x, r_1, \ldots, r_{q(|x|)} \rangle$, to the $\mathrm{NP}^{\mathrm{SAT} \oplus H}$ set

$$E' = \{\langle x, r_1, \ldots, r_z \rangle \mid z = q(|x|) \text{ and}$$
$$(\forall i : 1 \leq i \leq z)\,[|r_i| \leq p(i)] \text{ and } (\exists y : |y| \leq q(|x|))\,[y \in H \iff \langle y, r_{|y|} \rangle \notin S_1]\},$$

and if the answer is "no," we know the "r" advice collection is good. Similarly, with one question to an $\mathrm{NP}^{\mathrm{SAT} \oplus H}$ set E'' (defined analogously), we determine whether the "s" advice collection is good for helping S_2.

Note that *when given the correct advice strings*, an NP machine can strongly (in the sense of Long [26] and Selman [35]) check whether $x \in H$ or $x \notin H$ by nondeterministically guessing which is true, checking an $x \in H$ guess via checking whether $\langle x, r_{|x|} \rangle \in S_1$, and checking an $x \notin H$ guess via checking whether $\langle x, s_{|x|} \rangle \in S_2$.

Our simulation of $B = L(N_1^{L(N_2^{L(N_3^H)})})$ in $\mathrm{NP}^{\mathrm{NP}^{\mathrm{SAT} \oplus H}}$ proceeds as follows (for simplicity, we will call our base machine N_4). N_4 guesses and checks good advice sets as already described. N_4 now simulates N_1, except each time N_1 asks a query y to $L(N_2^{L(N_3^H)})$, N_4 asks the query $\langle y, \langle r_1, \ldots, r_{q(|x|)} \rangle, \langle s_1, \ldots, s_{q(|x|)} \rangle \rangle$ to an $\mathrm{NP}^{\mathrm{SAT} \oplus H}$ set E''', which itself will satisfy $E''' = L(N_5^{\mathrm{SAT} \oplus H})$ for a machine N_5 to be defined. (Since we have only one $\mathrm{NP}^{\mathrm{SAT} \oplus H}$ oracle, the actual set we will use is $E = E' \oplus E'' \oplus E'''$.) N_5 on input $\langle y, \langle r_1, \ldots, r_t \rangle, \langle s_1, \ldots, s_t \rangle \rangle$ simulates N_2 on input y, except every time N_2 asks a query z to $L(N_3)$ on input y, N_5 asks the query $\langle z, \langle r_1, \ldots, r_t \rangle, \langle s_1, \ldots, s_t \rangle \rangle$ to the NP set G (since SAT is NP-complete, we implicitly convert the query to an appropriate query to SAT):

> $G = \{\langle z, \langle r_1, \ldots, r_t \rangle, \langle s_1, \ldots, s_t \rangle \rangle \mid$ if we simulate $N_3^H(z)$, replacing each call to H (say "$w \in H$?") by nondeterministically checking whether $\langle w, r_{|w|} \rangle \in S_1$ (in which case we proceed along the path certifying $\langle w, r_{|x|} \rangle$ as of $w \in H$) and (separately, nondeterministically) whether $\langle w, s_{|w|} \rangle \notin S_2$ (in which case we proceed as if $w \notin H$), we have an accepting path of our simulated $N_3\}$. *Note:* If any of the w are such that $|w| > t$, we act as if $s_{|w|} = r_{|w|} = \epsilon$, since in actual runs this case will not occur.

We make no claim that $G \in \mathrm{NP} \bigcap \mathrm{coNP}$. In fact, with "bad" advice as inputs, the simulation defining G will be quite chaotic: a query "$w \in H$?" might be treated as being answered both "yes" and "no" or neither "yes" nor "no." However, *when given good advice sets*, the machine will in fact correctly simulate $N_3^H(z)$: each query w of $N_3(z)$ will be answered either "yes" or "no," will not be answered both "yes" and "no," and will be answered correctly. That is, G's simulation of H is an example of strong computation *when the advice is correct.* Crucially, for every query actually asked of G during an actual run of our $\mathrm{NP}^{E' \oplus E'' \oplus L(N_5^G)}$ algorithm, the advice will be correct (and thus the strong computation going on within G will be correct). Recall that this behavior, in which every *actual access* to an oracle maintains a certain nice property of the oracle computation (such as computing strongly), though some queries that are never asked might taint the property, is known as "guarded" access. We have now given an $\mathrm{NP}^{\mathrm{NP}^{\mathrm{SAT}\oplus H}}$ simulation of an arbitrary set $B \in \mathrm{NP}^{\mathrm{NP}^{\mathrm{NP}^H}}$ for arbitrary $H \in$ (NP/poly) $\bigcap$ (coNP/poly). □

4. NPMV-selectivity versus self-reducibility.

Buhrman, van Helden, and Torenvliet [8] showed that if a self-reducible set is P-selective, then it is in P, and Hemaspaandra et al. [18] proved that if a self-reducible set is NPMV_t-selective, then it is in NP $\cap$ coNP. We prove, as Theorem 4.3 below, a similar result for self-reducible NPMV-selective sets and apply this result to PSPACE and the levels of the polynomial hierarchy.

The standard definition of self-reducibility that is used in most contemporary research in complexity theory was given by Meyer and Paterson [29].

DEFINITION 4.1. *A polynomial time computable partial order $<$ on Σ^* is* OK *if and only if*

1. *each strictly decreasing chain is finite and there is a polynomial p such that every finite $<$-decreasing chain is shorter than p of the length of its maximum element, and*
2. *for all $x, y \in \Sigma^*$, $x < y$ implies that $|x| \leq p(|y|)$.*

DEFINITION 4.2. *A set L is* self-reducible *if there is an OK partial order $<$ and a deterministic polynomial time-bounded machine M such that M accepts L with oracle L and, on any input x, M asks its oracle only about words strictly less than x in the OK partial order $<$. If the self-reduction of the query machine M in fact is also a polynomial-time disjunctive (conjunctive) truth-table reduction, then L is disjunctive (conjunctive) self-reducible.*

Note in particular that unless otherwise specified, we use self-reducible to mean Turing self-reducible.

THEOREM 4.3. *If A is self-reducible and* NPMV*-selective, then $A \in$ NP.*

Proof. First, we need to introduce some notation. Let B be a set and S be a finite set. Let $\succeq$ be a total order over S such that for every $x, y \in S$, $x \succeq y \iff (x \in B \Rightarrow y \in B)$. Then for each $x, y \in S$, define $x \equiv y$ if there exist some $w_1, \ldots, w_m \in S$ such that (i) both x and y appear in $w_1, \ldots, w_m$, (ii) $w_m = w_1$, and (iii) for every i, $1 \leq i \leq m-1$, $w_i \succeq w_{i+1}$, and define $x \succ y$ if $x \succeq y$ and $x \not\equiv y$. Call $x \in S$ *minimal* if for every $y \in S$, either $x \equiv y$ or $x \succ y$. Note that $x \equiv y$ implies $x \in B$ if and only if $y \in B$, and therefore, for any minimal x, $x \in B$ implies $S \subseteq B$. Also note that finding all minimal elements in S is equivalent to dividing a "directed clique" (i.e., a clique in which each edge is directed) into its fully connected components and finding the (necessarily unique) component from which no other component is reachable. So, after knowing whether $x \succeq y$ or $y \succeq x$ for each $x, y \in S$, finding all minimal elements can be done in time polynomial in $\sum_{x \in S} |x|$.

Let A be self-reducible via a machine M and an OK partial order $<$ as in Definition 4.2. Let $\perp$ be a fixed element in A and, without loss of generality, assume for every $x \in \Sigma^*$ other than $\perp$ that $\perp < x$. Let A be NPMV-selective with selector function $f \in$ NPMV. Consider the nondeterministic Turing machine N defined, on input x, by the following algorithm.

(1) Nondeterministically guess one computation path of M on x together with oracle answers and put into S_1 (S_0) all the queries for which affirmative (negative) answers are guessed.
If M on x along the guessed path rejects, then reject x.

(2) For each $y \in S_0$ and $z \in S_1 \cup \{x\}$, nondeterministically verify that $z \in \textit{set-}f(y, z)$.
If the verification is not successful for some y, z, then reject x.

(3) For each $y, z \in S_1$, nondeterministically compute $f(y, z)$ and define $y \succeq z$ if $f(y, z) = z$ and $z \succeq y$ if $f(y, z) = y$.
If for some y and z, computing $f(y, z)$ is not successful, then reject x.

(4) If $S_1 = \emptyset$, then output $\perp$. Otherwise, output lexicographically the smallest minimal string in S_1.

It is easy to see that N is polynomial-time bounded. We claim the following:

- For every $x \notin A$, if N on x outputs y, then $y < x$ and $y \notin A$.
- For every $x \in A$,
  - N on x has an output in A and
  - every output y of N on x satisfies $(y = \perp) \vee (y < x)$.

This is seen as follows. Suppose that $x \notin A$ and N on x outputs w at step (4). Since N must choose an accepting computation path of M on x, either $S_0 \not\subseteq \overline{A}$ or $S_1 \not\subseteq A$. However, the former is not the case because, since the verifications in step (2) are all successful, S_0 having an element in A implies $x \in A$, yielding a contradiction. So the latter is the case. Since w is minimal in S_1, $w \in A$ implies $S_1 \subseteq A$. So w cannot be in A. Hence the first claim holds.

On the other hand, suppose that $x \in A$. The machine N can guess the "correct" accepting computation path of M on x for which $S_0 \subseteq \overline{A}$ and $S_1 \subseteq A$. After guessing the path, N can reach step (4) because for every $y \in S_0$ and $z \in S_1$, $\textit{set-}f(y, z) = \{z\}$, and for every $y, z \in S_1$, $\textit{set-}f(y, z) \neq \emptyset$. After entering step (4), N will choose its output from $\{\perp\} \cup S_1$, which is a subset of A. So N will output a string in A. Hence the second claim holds.

Now consider a machine D that, on input x, starting with $w = x$, executes the following algorithm.

(I) Simulate N on w.

(II) If N rejects, then reject. If N outputs $\perp$, then accept. Otherwise, set w to the output of N and go back to (I)

By the definition of self-reducibility, step (I) is repeated at most polynomially many times, and thus D is polynomial-time bounded. By the above two claims, if $x \notin A$, then D never obtains $\perp$ as the output of N, and if $x \in A$, for some computation path, D obtains $\perp$ as the output of N. So D accepts x if and only if $x \in A$. This establishes that $A \in \mathrm{NP}$. □

Note that from the well-known fact that every disjunctive self-reducible set is in NP and from the fact that every set in NP is NPMV-selective, it follows that every disjunctive self-reducible set is NPMV-selective. Theorem 4.3 yields, for example, the following consequences, keeping in mind the fact that PSPACE and each Σ_k^p have self-reducible complete sets. Note that the $k \geq 2$ below cannot be improved to $k \geq 1$ unless PH = NP.

COROLLARY 4.4.

1. PSPACE $\subseteq$ NPSV-sel *if and only if* PSPACE $\subseteq$ NPMV-sel *if and only if* PSPACE = NP.
2. *For any* $k \geq 2$, $\Sigma_k^p \subseteq$ NPSV-sel *if and only if* $\Sigma_k^p \subseteq$ NPMV-sel *if and only if* PH = NP.
3. coNP $\subseteq$ NPMV-sel *if and only if* NP = coNP.

5. Conclusion and open questions. This paper has studied the complexity of computing a *single* satisfying assignment of an input satisfiable formula. Previously, it was (trivially) known that satisfying assignments could be found by polynomial-time functions if and only if P = NP. It was also (trivially) known that an assignment could be found via a polynomial-time machine using an NP oracle (and it was known, not trivially, that finding the lexicographically largest assignment was the hardest of all problems solvable in that class [25]).

But what about function classes between FP and $\mathrm{FP}^{\mathrm{NP}}$? In this paper, we proved that the NPSV functions are unlikely to have the power to find satisfying assignments; they can do so only if the polynomial hierarchy collapses to $\mathrm{ZPP}^{\mathrm{NP}}$. There remains an important function class intermediate in power between the NPSV functions (shown in this paper to be unlikely to have the power of finding satisfying assignments) and the functions computable via Turing access to an NP oracle (which clearly can find satisfying assignments). This class is the class of (partial) functions computable via *parallel* (that is, truth-table) access to an NP oracle.

Clearly, NPSV is a subset of this class (cf. [38]). The key open issue is distilled in the following hypothesis (see [43, 1, 17, 31, 38] for background and discussion).

HYPOTHESIS 5.1. *Every* NPMV *function has a (single-valued) refinement in* $\mathrm{FP}_{tt}^{\mathrm{NP}}$.

The above can be equivalently phrased as: There is a partial function f computable by a polynomial-time Turing machine making parallel queries to NP such that for each formula $F \in$ SAT, $f(F)$ is a satisfying assignment of F. Does Hypothesis 5.1 imply a collapse of the polynomial hierarchy? It seems that such a result would require techniques substantially different from those of this paper. In particular, our result that "NPMV has NPSV refinements only if $\mathrm{ZPP}^{\mathrm{NP}} = \mathrm{PH}$" itself relativizes. However, any relativizable proof of "Hypothesis 5.1 implies a collapse of the polynomial hierarchy" would immediately imply—due to the result of Watanabe and Toda [43] that Hypothesis 5.1 holds relative to a random oracle—that the polynomial hierarchy collapses relative to a random oracle. Furthermore, if the polynomial hierarchy collapses relative to a random oracle, then the polynomial hierarchy collapses (see [5]; see also [32]). The main result of the present paper does not similarly imply that the polynomial hierarchy collapses relative to a random oracle, since Hypothesis 1.1 is known to fail relative to a random oracle [33].

Acknowledgments. The authors would like to thank S. Biswas, H. Buhrman, L. Fortnow, Y. Han, E. Hemaspaandra, and M. Zimand for many helpful comments and suggestions.

REFERENCES

[1] K. ABRAHAMSON, M. FELLOWS, AND C. WILSON, *Parallel self-reducibility*, in Proc. 4th International Conference on Computing and Information, IEEE Computer Society Press, Los Alamitos, CA, 1992, pp. 67–70.

[2] E. ALLENDER, *Invertible functions*, Ph.D. thesis, Georgia Institute of Technology, Atlanta, 1985.

[3] E. ALLENDER AND L. HEMACHANDRA, *Lower bounds for the low hierarchy*, J. Assoc. Comput. Mach., 39 (1992), pp. 234–251.

[4] J. BALCÁZAR, R. BOOK, AND U. SCHÖNING, *Sparse sets, lowness and highness*, SIAM J. Comput., 15 (1986), pp. 739–746.

[5] R. BOOK, *On collapsing the polynomial-time hierarchy*, Inform. Process. Lett., 52 (1994), pp. 235–237.

[6] R. BOOK, T. LONG, AND A. SELMAN, *Quantitative relativizations of complexity classes*, SIAM J. Comput., 13 (1984), pp. 461–487.

[7] H. BUHRMAN, J. KADIN, AND T. THIERAUF, *On functions computable with nonadaptive queries to NP*, in Proc. 9th Structure in Complexity Theory Conference, IEEE Computer Society Press, Los Alamitos, CA, 1994, pp. 43–52.

[8] 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, Los Alamitos, CA, 1993, pp. 44–51.

[9] D. DENNY-BROWN, Y. HAN, L. HEMASPAANDRA, AND L. TORENVLIET, *Semi-membership algorithms: Some recent advances*, SIGACT News, 25 (1994), pp. 12–23.

[10] S. EVEN AND Y. YACOBI, *Cryptocomplexity and NP-completeness*, in Proc. 7th International Colloquium on Automata, Languages, and Programming, Lecture Notes in Comput. Sci., Springer-Verlag, Berlin, New York, Heidelberg, 1980, pp. 195–207.

[11] S. FENNER, L. FORTNOW, S. KURTZ, AND L. LI, *An oracle builder's toolkit*, in Proc. 8th Structure in Complexity Theory Conference, IEEE Computer Society Press, Los Alamitos, CA, 1993, pp. 120–131.

[12] S. FENNER, S. HOMER, M. OGIWARA, AND A. SELMAN, *On using oracles that compute values*, in Proc. 10th Annual Symposium on Theoretical Aspects of Computer Science, Lecture Notes in Comput. Sci. 665, Springer-Verlag, Berlin, New York, Heidelberg, 1993, pp. 398–407.

[13] R. GAVALDÀ, *Bounding the complexity of advice functions*, in Proc. 7th Structure in Complexity Theory Conference, IEEE Computer Society Press, Los Alamitos, CA, 1992, pp. 249–254.

[14] R. GAVALDÀ AND J. BALCÁZAR, *Strong and robustly strong polynomial time reducibilities to sparse sets*, Theoret. Comput. Sci., 88 (1991), pp. 1–14.

[15] J. GROLLMANN AND A. SELMAN, *Complexity measures for public-key cryptosystems*, SIAM J. Comput., 17 (1988), pp. 309–335.

[16] Y. HAN, L. HEMASPAANDRA, AND T. THIERAUF, *Threshold computation and cryptographic security*, in Proc. 4th International Symposium on Algorithms and Computation, Lecture Notes in Comput. Sci. 762, Springer-Verlag, Berlin, New York, Heidelberg, 1993, pp. 230–239.

[17] E. Hemaspaandra, A. Naik, M. Ogiwara, and A. Selman, *P-selective sets, and reducing search to decision vs. self-reducibility*, Tech. report 93-21, Department of Computer Science, State University of New York at Buffalo, Buffalo, NY, 1993.

[18] L. Hemaspaandra, A. Hoene, A. Naik, M. Ogiwara, A. Selman, T. Thierauf, and J. Wang, *Selectivity: Reductions, nondeterminism, and function classes*, Tech. report TR-469, Department of Computer Science, University of Rochester, Rochester, NY, 1993.

[19] L. Hemaspaandra, 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, Los Alamitos, CA, 1993, pp. 55–59.

[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 ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1980, pp. 302–309; an extended version has also appeared as *Turing machines that take advice*, 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] J. Köbler, *Locating P/poly optimally in the extended low hierarchy*, Theoret. Comput. Sci., 134 (1994), pp. 263–285.

[24] J. Köbler and O. Watanabe, *New collapse consequences of NP having small circuits*, Tech. report 94-11, Institut für Informatik, Universität Ulm, Ulm, Germany, 1994.

[25] M. Krentel, *The complexity of optimization problems*, J. Comput. System Sci., 36 (1988), pp. 490–509.

[26] T. Long, *Strong nondeterministic polynomial-time reducibilities*, Theoret. Comput. Sci., 21 (1982), pp. 1–25.

[27] T. Long and M. Sheu, *A refinement of the low and high hierarchies*, Tech. report OSU-CISRC-2/91-TR6, Department of Computer Science, Ohio State University, Columbus, OH, 1991.

[28] L. Longpré and A. Selman, *Hard promise problems and nonuniform complexity*, Theoret. Comput. Sci., 115 (1993), pp. 277–290.

[29] A. Meyer and M. Paterson, *With what frequency are apparently intractable problems difficult?*, Tech. report MIT/LCS/TM-126, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 1979.

[30] A. Meyer and L. Stockmeyer, *The equivalence problem for regular expressions with squaring requires exponential space*, in Proc. 13th IEEE Symposium on Switching and Automata Theory, IEEE Press, Piscataway, NJ, 1972, pp. 125–129.

[31] A. Naik, M. Ogiwara, and A. Selman, *P-selective sets, and reducing search to decision vs. self-reducibility*, in Proc. 8th Structure in Complexity Theory Conference, IEEE Computer Society Press, Los Alamitos, CA, 1993, pp. 52–64.

[32] N. Nisan and A. Wigderson, *Hardness vs. randomness*, J. Comput. System Sci., 49 (1994), pp. 149–167.

[33] J. Royer, personal communication, Aug. 1993.

[34] U. Schöning, *A low and a high hierarchy within NP*, J. Comput. System Sci., 27 (1983), pp. 14–28.

[35] A. Selman, *Polynomial time enumeration reducibility*, SIAM J. Comput., 7 (1978), pp. 440–457.

[36] ———, *P-selective sets, tally languages, and the behavior of polynomial time reducibilities on NP*, Math. Systems Theory, 13 (1979), pp. 55–65.

[37] ———, *Promise problems complete for complexity classes*, Inform. Comput., 78 (1988), pp. 87–98.

[38] ———, *A taxonomy of complexity classes of functions*, J. Comput. System Sci., 48 (1994), pp. 357–381.

[39] L. Stockmeyer, *The polynomial-time hierarchy*, Theoret. Comput. Sci., 3 (1977), pp. 1–22.

[40] S. Toda, *PP is as hard as the polynomial-time hierarchy*, SIAM J. Comput., 20 (1991), pp. 865–877.

[41] S. Toda and M. Ogiwara, *Counting classes are at least as hard as the polynomial-time hierarchy*, SIAM J. Comput., 21 (1992), pp. 316–328.

[42] L. Valiant and V. Vazirani, *NP is as easy as detecting unique solutions*, Theoret. Comput. Sci., 47 (1986), pp. 85–93.

[43] O. Watanabe and S. Toda, *Structural analysis of the complexity of inverse functions*, Math., Systems Theory, 26 (1993), pp. 203–214.

[44] C. Yap, *Some consequences of non-uniform conditions on uniform classes*, Theoret. Comput. Sci., 26 (1983), pp. 287–300.

SIAM J. COMPUT.
Vol. 25, No. 4, pp. 709–739, August 1996

002

A METHOD OF CONSTRUCTING SELECTION NETWORKS WITH $O(\log n)$ DEPTH*

S. JIMBO† AND A. MARUOKA†

Abstract. A classifier with n inputs is a comparator network that classifies a set of n values into two classes with the same number of values in such a way that each value in one class is at least as large as all of those in the other. Based on the utilization of expanders, Pippenger constructed classifiers with n inputs whose size is asymptotic to $2n \log_2 n$. In the same spirit, we obtain a relatively simple method of constructing classifiers of depth $O(\log n)$. Consequently, for an arbitrary constant $C > 3/\log_2 3 = 1.8927\ldots$, we construct classifiers of depth $O(\log n)$ and of size at most $Cn \log_2 n + O(n)$.

Key words. comparator network, selection network, classifier, expander

AMS subject classifications. 68Q22, 94C10

1. Introduction. A classifier with n inputs is a comparator network (see Knuth [5]) that classifies a set of n values into two classes with the same number of values in such a way that each value in one class is at least as large as all of those in the other class, whereas an n-sorter is a comparator network that sorts n values into order. Since an n-sorter is obviously a classifier with n inputs, the existence of n-sorters of size $O(n \log n)$ and depth $O(\log n)$, given in [1] and [2], immediately implies the existence of classifiers of the same size and depth. It seems that the main concern in [1] and [2] is to prove just the existence of n-sorters of size $O(n \log n)$ and depth $O(\log n)$, so the construction of the sorters is of some intricacy, and the constant factors in the depth and size bound are enormous. As a result of further efforts to improve the previous construction, it is shown in [6] that there is a family of sorting networks of depth smaller than $6100 \log_2 n$, but the numerical constant in the size bound has not been brought below 1000. Since classifiers do not do as much as sorters, it is natural to ask whether we can obtain much simpler construction of classifiers with smaller constant factors. Based on the same fundamental idea—namely the utilization of expanders—as their sorters, Pippenger [7] constructed classifiers with size asymptotic to $2n \log_2 n$ and $O(\log^2 n)$ depth. Although he noticed without giving any explicit construction that his construction could be modified to provide a bound of $O(\log n)$ for depth, he also mentioned that his method does not seem well suited to optimize the depth. In the same spirit, we investigate the efficient construction of classifiers, obtaining relatively simple classifiers of depth $O(\log n)$. In consequence, for arbitrary $C > 3/\log_2 3 = 1.8927\ldots$, we can construct a family of classifiers of depth $O(\log n)$ and of size at most $Cn \log_2 n + O(n)$, which improves Pippenger's construction by the constant factor in size and at the same time by order in depth.

Unlike the recursive construction due to Pippenger, our classifier is constructed by connecting $O(\log n)$ modules called layers in cascade and then removing unnecessary submodules of layers called compressors. Although the depth of some of the layers is $\Omega(\log n)$, it turns out that each layer contributes only a constant amount to the depth in the total construction, so that after all the depth of the classifier becomes $O(\log n)$.

We employ asymmetric expanders as basic components to construct various modules, which in turn work as building blocks to construct the layers. This is contrasts with the construction, due to Pippenger, based on symmetric expanders, where an expander is called symmetric if the number of its left vertices is equal to that of its right vertices and asymmetric otherwise. Allowing various types of expanders helps us to obtain suitable modules and hence to get more efficient classifiers.

*Received by the editors April 30, 1993; accepted for publication (in revised form) November 28, 1994.

†Graduate School of Information Sciences, Tohoku University, Sendai 980-77, Japan (jimbo, maruoka@ecei.tohoku.ac.jp).

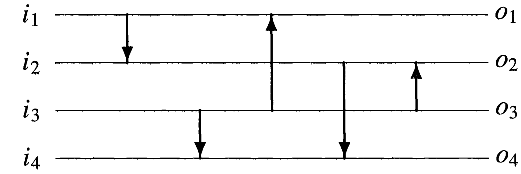


FIG. 1. *Representation of a comparator network.*

In §2, we give, together with some basic lemmas used frequently later, definitions of comparator subnetworks, called a compressor and an extractor, both of which are constructed from expanders. In §3, after giving a definition of a layer, we define a comparator network, denoted $\mathcal{N}^n$, consisting of modules such as compressors, extractors, and sorters. In §4, we analyze the behavior of the comparator network $\mathcal{N}^n$ and verify that the comparator network works as a classifier. We also establish a bound of $Cn\log_2 n + O(n)$ on the size of the classifier and a bound of $O(\log n)$ on its depth, where C is any constant greater than $3/\log_2 3$. Finally, in §5, we present conclusions mentioning some generalizations of the result obtained in the present paper.

2. Preliminaries. As in Knuth [5], a comparator network is given as a network illustrated in Fig. 1.

Each comparator is represented by a vertical connection with an arrow between two horizontal lines called registers. Input values enter at the left and move along the registers. Each comparator causes an interchange of its inputs if necessary, so that the larger value appears on the register on the head side of the comparator after passing the comparator. For a comparator in a comparator network, we call the register on the head side of the comparator the *head register* and the register on the tail side of the comparator the *tail register*.

Let N be a comparator network. The size of N, denoted by size(N), is the number of comparators belonging to N. The depth of N, denoted by depth(N), is the maximum number of comparators on a path from an input terminal of N to an output terminal of N which passes along the registers, from the left to the right, and comparators.

When we consider a comparator network N, we always take a one-to-one correspondence ρ_N from the registers of N onto the set $\{1, 2, \ldots, n\}$, where n is the number of the registers of N. We may later identify a register r of N with an integer $\rho_N(r)$ if it causes no confusion.

DEFINITION 2.1 (a classifier). *Let n be an even positive integer. Let S be a totally ordered set. A* classifier *is a comparator network with n registers and n/2 designated output terminals such that each value appearing on the designated output terminals is at most as large as any value appearing on the remaining output terminals for any input in* S^n.

By a method similar to the proof of the zero–one principle (in [5]), the following statement can be proved: If, when given n binary values, arbitrarily on its input terminals, a comparator network always selects (as the values on k designated output terminals) the smallest k values among the n binary values, then, it will always select the smallest k values among arbitrary n values in an arbitrary totally ordered set S. Hence, without loss of generality, we assume that the input set to classifiers is $\{0, 1\}^n$ rather than S^n.

For x in $\{0, 1\}^n$, $\#_1 x$ denotes the number of x's components equal to 1 and $\#_0 x$ denotes the number of x's components equal to 0. Note that any Boolean function implemented by a comparator network is monotone in the sense described as follows: Let x and x' be vectors in $\{0, 1\}^n$, and let y and y' denote the outputs of N corresponding to x and x', respectively; then $x \le x'$ implies $y \le y'$, where $(a_1, \ldots, a_n) \le (b_1, \ldots, b_n)$ means that $a_1 \le b_1, \ldots, a_{n-1} \le b_{n-1}$ and $a_n \le b_n$.

DEFINITION 2.2 (an expander). *Let* $0 < \alpha \leq 1$ *and* $\beta \geq 1$. *Let* $G = (A, B, E)$ *be a bipartite graph with left vertex set* A, *right vertex set* B *and edge set* E. G *is an* (α, β)-expander *if* $|\{y \in B \mid (\exists x \in X)((x, y) \in E)\}| \geq \beta|X|$ *holds for every subset* $X \subseteq A$ *with* $|X| \leq \alpha|A|$.

DEFINITION 2.3 (a compressor). *Let* n *and* m *be positive integers and let* α *and* β *be real numbers with* $0 < \alpha < 1$ *and* $\beta > 1$. *Let* $x = (x_1, x_2, \ldots, x_{n+m})$ *be an input to a comparator network with* $n + m$ *registers, and let* $y = (y_1, y_2, \ldots, y_{n+m})$ *denote the output corresponding to* x. *An* (n, m, α, β)-compressor of type 1 *is a comparator network with* $n + m$ *registers such that if* x *is in* $\{0, 1\}^{n+m}$ *and* $\#_1 x \leq \lceil(\beta + 1)\lfloor\alpha n\rfloor\rceil$, *then* $\beta\#_1(y_1, y_2, \ldots, y_n) \leq \#_1(y_{n+1}, y_{n+2}, \ldots, y_{n+m})$. *An* (n, m, α, β)-compressor of type 0 *is a comparator network with* $n + m$ *registers such that if* x *is in* $\{0, 1\}^{n+m}$ *and* $\#_0 x \leq \lceil(\beta + 1)\lfloor\alpha n\rfloor\rceil$, *then* $\beta\#_0(y_{m+1}, y_{m+2}, \ldots, y_{n+m}) \leq \#_0(y_1, y_2, \ldots, y_m)$. *The registers* $1, 2, \ldots, n$ *of an* (n, m, α, β)*-compressor of type* 1 *and the registers* $1, 2, \ldots, m$ *of an* (n, m, α, β)*-compressor of type* 0 *are called* upper registers. *The registers* $n + 1, n + 2, \ldots, n + m$ *of an* (n, m, α, β)*-compressor of type* 1 *and the registers* $m + 1, m + 2, \ldots, n + m$ *of an* (n, m, α, β)*-compressor of type* 0 *are called* lower registers.

DEFINITION 2.4 (an extractor). *Let* n *and* m *be positive integers, and let* γ *be a real number with* $0 \leq \gamma \leq 1$. *An* (n, m, γ)-extractor *is a comparator network* N *with* $n + 2m$ *registers such that for every* $x = (x_1, \ldots, x_{n+2m})$ *in* $\{0, 1\}^{n+2m}$, $\#_1 x \leq n + m$ *implies* $\#_1(y_1, \ldots, y_m) \leq \gamma m$ *and* $\#_0 x \leq n + m$ *implies* $\#_0(y_{n+m+1}, \ldots, y_{n+2m}) \leq \gamma m$, *where* $(y_1, \ldots, y_{n+2m})$ *is the output of* N *corresponding to* x. *The registers* $1, 2, \ldots, m$, *the registers* $m + 1, m + 2, \ldots, n + m$ *and the registers* $n + m + 1, n + m + 2, \ldots, n + 2m$ *of an* (n, m, γ)*-extractor are called* upper registers, middle registers, *and* lower registers, *respectively.*

It is easy to see that an (n, m, α, β)-compressor of type 1 becomes an (n, m, α, β)-compressor of type 0 by interchanging register 1 and register $n + m$, register 2 and register $n + m - 1$, and so on, and keeping the direction of each comparator.

PROPOSITION 2.5 (see [1], [2]). *Assume that there exists an* (α, β)*-expander with* n *left vertices and* m *right vertices. Let* s *denote the number of edges and* k *the maximum degree of a vertex of the expander. Then there exist* (n, m, α, β)*-compressors of types both* 1 *and* 0 *of size at most* s *and of depth at most* k.

Proof. We only prove the existence of an (n, m, α, β)-compressor of type 1. The existence of an (n, m, α, β)-compressor of type 0 is proved in a similar way.

Let $G = (U, V, E)$ denote the (α, β)-expander, where $U = \{u_1, \ldots, u_n\}$ is the set of the left vertices of G, $V = \{v_1, \ldots, v_m\}$ is the set of the right vertices of G, and $E \subseteq U \times V$ is the set of the edges of G. Since the maximum degree of G is k, we can color the edges of the expander with k colors in such a way that no two adjacent edges have the same color. Let $C_1, C_2, \ldots, C_k$ denote the k colors. From G along with the coloring above, we can construct a comparator network N that satisfies the following properties:

1. N has $n + m$ registers.

2. There is a one-to-one correspondence ρ from the edges of the expander onto the comparators of N. For every (u_i, v_j) in E, $\rho((u_i, v_j))$ is a comparator which connects the register i and the register $n + j$, and the head register of $\rho((u_i, v_j))$ is greater than the tail register of $\rho((u_i, v_j))$.

3. Let a and b be comparators incident to the same register. Let i and j be indices of colors such that C_i and C_j are the colors of $\rho^{-1}(a)$ and $\rho^{-1}(b)$, respectively. If a is between an input terminal and b on the register, then $i < j$ holds.

It is clear that size$(N) = s$ and depth$(N) = k$. The fact that N is an (n, m, α, β)-compressor of type 1 is proved as follows.

Let $x = (x_1, x_2, \ldots, x_{n+m})$ be an input to N, and let $y = (y_1, y_2, \ldots, y_{n+m})$ denote the output of N corresponding to x. Assume that $\#_1 x = \#_1 y \leq \lceil(\beta + 1)\lfloor\alpha n\rfloor\rceil$

and $\beta\#_1(y_1, y_2, \ldots, y_n) > \#_1(y_{n+1}, y_{n+2}, \ldots, y_{n+m})$. Let $I = \{u_i \in U \mid y_i = 1\}$ and $J = \{v_j \in V \mid y_{j+n} = 1\}$. In the case where $|I| \leq \alpha n$, the number of the right vertices of G which are adjacent to a left vertex in I is at least $\beta|I| > |J|$ by the assumption. In the case where $|I| > \alpha n$, the number of the right vertices of G which are adjacent to a left vertex in I is at least $\lceil\beta\lfloor\alpha n\rfloor\rceil > |J|$ because $|J| = \#_1 y - |I| < \lceil(\beta+1)\lfloor\alpha n\rfloor\rceil - \lfloor\alpha n\rfloor = \lceil\beta\lfloor\alpha n\rfloor\rceil$. In both cases, therefore, there exists an edge (u_μ, v_ν) in E such that u_μ is in I and v_ν is not in J. Hence there exists a comparator c of N which connects register μ and register ν. Since register μ is the tail register of any comparator incident to the register, a value on register μ never changes from 0 to 1 by passing a comparator. Therefore, the fact that the output of N on register μ is 1 (i.e., $u_\mu \in I$) implies that any value on register μ is 1, and hence the output of comparator c on register μ is 1. Similarly, the output of comparator c on register ν is 0. These facts contradict the fact that c is a comparator. □

Proposition 2.6 below is obtained by modifying Bassalygo's lemma (in [4]) and hence its proof. Although Proposition 2.6 is almost the same as Bassalygo's lemma in appearance, the former is stronger than the latter in some sense. This is because our definition of expanders is stronger than Bassalygo's in [4]. To prove Proposition 2.6, we followed the argument due to Bassalygo [4], correcting some careless calculation and adding calculations concerning ceiling and floor. The proof of the statement is given in the appendix.

PROPOSITION 2.6. *For any positive integers q and p, any positive real numbers α and β $(0 < \alpha < p/\beta q < 1)$, and a sufficiently large integer n $(n \geq n_0(\alpha, \beta, p, q))$, there exists an (α, β)-expander with qn left vertices and pn right vertices such that the number of edges is less than or equal to $spqn$, the maximum degree of the left vertices is less than or equal to sp, and the maximum degree of the right vertices is less than or equal to sq, where s is any integer greater than*

$$\max\left\{\frac{H(\alpha) + (p/q)H(\alpha\beta q/p)}{pH(\alpha) - \alpha\beta q H(p/\beta q)}, \frac{p(1+\beta) - \alpha\beta(p+q)}{p(p-\alpha\beta q)}\right\}.$$

In the expression above, $H(x)$ is defined to be $-x\log x - (1-x)\log(1-x)$ for $0 < x < 1$.

Lemmas 2.7 and 2.8 below are obtained directly from Proposition 2.6.

LEMMA 2.7. *Let $k \geq 2$ and $m \geq 1$ be integers. For every $\varepsilon > 0$ with $m(k-1-\varepsilon) > 1$, there exist an integer $n_0 > 0$ and a real number α with $0 < \alpha < 1$ such that for all positive integers $n \geq n_0$ and $l \leq mn$, there exists an $(mn\alpha/l, k-1-\varepsilon)$-expander with l left vertices and n right vertices such that the degree of every left vertex is at most k and the degree of every right vertex is at most mk.*

Proof. Let $p = 1, q = m, \beta = k-1-\varepsilon$, and $s = k$. Since

$$\lim_{\alpha\to 0}\max\left\{\frac{H(\alpha) + (p/q)H(\alpha\beta q/p)}{pH(\alpha) - \alpha\beta q H(p/\beta q)}, \frac{p(1+\beta) - \alpha\beta(p+q)}{p(p-\alpha\beta q)}\right\} = \frac{1+\beta}{p} = k - \varepsilon,$$

there exists a real number α such that

$$0 < \alpha < \frac{p}{\beta q} = \frac{1}{m(k-1-\varepsilon)} < 1$$

and

$$s = k > \max\left\{\frac{H(\alpha) + (p/q)H(\alpha\beta q/p)}{pH(\alpha) - \alpha\beta q H(p/\beta q)}, \frac{p(1+\beta) - \alpha\beta(p+q)}{p(p-\alpha\beta q)}\right\}.$$

Note that $x/H(x) \to 0 \quad (x \to 0)$ and, for every $c > 0$, $H(cx)/H(x) \to c \quad (x \to 0)$. By Proposition 2.6, therefore, there exists a positive integer n_0 such that, for any integer $n \geq n_0$, there exists an $(\alpha, k - 1 - \varepsilon)$-expander $G = (U, V, E)$ with mn left vertices and n right vertices such that the degree of every left vertex is at most k and the degree of every right vertex is at most mk. Let $U' \subseteq U$ be an arbitrary subset of left vertices of G with $|U'| = l$. By the definition of expanders, the induced subgraph $G' = (U', V, E')$ of G is an $(mn\alpha/l, k - 1 - \varepsilon)$-expander with l left vertices and n right vertices such that the degree of every left vertex is at most k and the degree of every right vertex is at most mk. □

LEMMA 2.8. *Let α and β be real numbers with $0 < \alpha < 1/\beta < 1$. There exist positive integers m_0 and k such that for all positive integers $m \geq m_0$ and $n \leq m$, there exists an $(m\alpha/n, \beta)$-expander with n left vertices and m right vertices such that the maximum degree of its vertices is at most k.*

Proof. Let $p = q = 1$ and

$$k = \left\lfloor \max\left\{ \frac{H(\alpha) + (p/q)H(\alpha\beta q/p)}{pH(\alpha) - \alpha\beta q H(p/\beta q)}, \ \frac{p(1+\beta) - \alpha\beta(p+q)}{p(p - \alpha\beta q)} \right\} \right\rfloor + 1.$$

Then by Proposition 2.6, there exists a positive integer m_0 such that, for any integer $m \geq m_0$, there exists an (α, β)-expander $G = (U, V, E)$ with m left vertices and m right vertices such that the maximum degree of its vertices is at most k. Let $U' \subseteq U$ be an arbitrary subset of left vertices of G with $|U'| = n$. By the definition of expanders, the induced subgraph $G' = (U', V, E')$ of G is an $(m\alpha/n, \beta)$-expander with n left vertices and m right vertices such that the maximum degree of its vertex is at most k. □

LEMMA 2.9. *For every $0 < \theta < 1$, every integer $k \geq 3$, and every $0 < \varepsilon < k - 2$, there exist a positive integer $n_A(\theta, k, \varepsilon)$ and a real number $\alpha_A(\theta, k, \varepsilon)$ with $0 < \alpha_A(\theta, k, \varepsilon) < 1$ such that, for all n and m with $n \geq n_A(\theta, k, \varepsilon)$ and $m \geq \theta n$, there exist an $(n, m, \alpha_A(\theta, k, \varepsilon), k - 1 - \varepsilon)$-compressor of type 1 and an $(n, m, \alpha_A(\theta, k, \varepsilon), k - 1 - \varepsilon)$-compressor of type 0 such that those depths are both at most $\lceil \theta^{-1} \rceil k$ and those sizes are both at most kn.*

Proof. Since $m \geq \theta n$ implies $n \leq \lceil \theta^{-1} \rceil m$, Lemma 2.9 follows from Proposition 2.5 and Lemma 2.7. □

LEMMA 2.10. *For every $0 < \gamma < 1$ and every $0 < \delta < 1/2$, there exist positive integers $n_B(\gamma, \delta)$ and $k_B(\gamma, \delta)$ such that for all positive integers m and n with $n \geq n_B(\gamma, \delta)$ and $m \geq \delta(m + n)$, there exists an (n, m, γ)-extractor of depth at most $k_B(\gamma, \delta)$ and size at most $k_B(\gamma, \delta)m$.*

Proof. Let $\beta = \frac{1-\delta\gamma}{\delta\gamma}$ and $\alpha = \frac{(2-\delta\gamma)\delta\gamma}{2(1-\delta\gamma)}$. Then $0 < \alpha < 1/\beta < 1$ holds. By Lemma 2.8, there exist positive integers m_0 and k such that, for all positive integers n and m with $n + m \geq m_0$, there exists an $(\alpha(m + n)/m, \beta)$-expander with m left vertices and $m + n$ right vertices such that the maximum degree of its vertex is at most k. Let $n_B(\gamma, \delta)$ and $k_B(\gamma, \delta)$ be $\max\{\lceil (1 - \delta)m_0 \rceil, \lceil \frac{2(1-\delta\gamma)}{\delta^2\gamma^2} \rceil\}$ and $2k$, respectively. It is easy to see that $\delta(m + n) \leq m$ and $n \geq n_B(\gamma, \delta)$ imply $m + n \geq m_0$. Thus by Proposition 2.5, for all positive integers m and n with $\delta(m + n) \leq m$ and $n \geq n_B(\gamma, \delta)$, there exist an $(m, m + n, \alpha(m + n)/m, \beta)$-compressor of type 1, say N_1, and an $(m, m + n, \alpha(m + n)/m, \beta)$-compressor of type 0, say N_0, such that $\text{size}(N_1) \leq km$, $\text{size}(N_0) \leq km$, $\text{depth}(N_1) \leq k$, and $\text{depth}(N_0) \leq k$. Let N denote the comparator network constructed by joining each output terminal of N_1 to the input terminal of N_0 of the same register number. Thus $\text{depth}(N) \leq k_B(\gamma, \delta)$ and $\text{size}(N) \leq k_B(\gamma, \delta)m$.

It will be proved that N is an (n, m, γ)-extractor. Let $x = (x_1, \ldots, x_{n+2m})$ be a vector in $\{0, 1\}^{n+2m}$, which is an input to N. Let $y = (y_1, \ldots, y_{n+2m})$ denote the output of N_1 corresponding to x and $z = (z_1, \ldots, z_{n+2m})$ the output of N_0 corresponding to y. That is, z is the output of N corresponding to x. Then the following inequality holds.

$$\begin{aligned}\left\lceil (\beta+1)\left\lfloor \frac{\alpha(m+n)}{m}\cdot m \right\rfloor \right\rceil &= \lceil (\beta+1)\lfloor \alpha(m+n)\rfloor \rceil \\ &\geq (\beta+1)((m+n)\alpha-1) \\ &= m+n+\frac{\delta\gamma}{2(1-\delta\gamma)}(m+n)-\frac{1}{\delta\gamma} \\ &\geq m+n+\frac{\delta\gamma}{2(1-\delta\gamma)}n_B(\gamma,\delta)-\frac{1}{\delta\gamma} \\ &\geq m+n+\frac{\delta\gamma}{2(1-\delta\gamma)}\left(\frac{2(1-\delta\gamma)}{\delta^2\gamma^2}\right)-\frac{1}{\delta\gamma}=m+n.\end{aligned}$$

First, assume that $\#_1 x \leq m+n$. Since N_1 is an $(m, m+n, \alpha(m+n)/m, \beta)$-compressor of type 1, $\beta\#_1(y_1,\dots,y_m) \leq \#_1(y_{m+1},\dots,y_{n+2m})$. Since $\#_1 x = \#_1 y \leq m+n$,

$$\#_1(y_1,\dots,y_m) \leq \frac{m+n}{1+\beta} \leq \frac{m}{(1+\beta)\delta} = \gamma m$$

holds. Since the tail register of each comparator of N incident to a register among $\{1,\dots,m\}$ belongs to $\{1,\dots,m\}$, we have

$$\#_1(z_1,\dots,z_m) \leq \#_1(y_1,\dots,y_m) \leq \gamma m.$$

Next, assume that $\#_0 x = \#_0 y \leq m+n$. Since N_0 is an $(m, m+n, \alpha(m+n)/m, \beta)$-compressor of type 0, $\beta\#_0(z_{m+n+1},\dots,z_{n+2m}) \leq \#_0(z_1,\dots,z_{m+n})$. We therefore have

$$\#_0(z_{m+n+1},\dots,z_{n+2m}) \leq \frac{m+n}{1+\beta} \leq \frac{m}{(1+\beta)\delta} = \gamma m. \qquad \square$$

Note that all the compressors and extractors constructed by Lemmas 2.9 and 2.10 are all of constant depth.

In the following sections, we construct the classifiers by connecting modules, such as compressors and extractors, repeatedly in cascade. To verify that the comparator network obtained in this way works as a classifier, we need to show that the error in the networks tends to decrease. Assuming some input vector is fed to the input terminals, we consider as the error the number of values appearing in the wrong output terminals. The next two lemmas indicate that, under certain conditions about the connection of comparators, the network's ability to decrease the error can be expressed in terms of that of the modules which constitute the whole network.

LEMMA 2.11. *Let N be a comparator network with n registers. Let i, j, and k be integers with $1 \leq i < j < k \leq n+1$. Let a and c be real constants with $a > 0$ and $0 \leq c \leq 1$. Assume that N satisfies the following conditions*:

1. *There is no comparator that connects a register in $\{1, 2, \dots, i-1\}$ and a register in $\{i, i+1, \dots, n\}$ and there is no comparator that connects a register in $\{1, 2, \dots, k-1\}$ and a register in $\{k, k+1, \dots, n\}$.*

2. *Let N' denote the comparator subnetwork of N composed of registers $i, i+1, \dots, k-1$ and the associated comparators of N. For every z in $\{0,1\}^{k-i}$, $\#_1 z \leq a$ implies $\#_1(N'_i(z),\dots,N'_{j-1}(z)) \leq c\#_1(N'_i(z),\dots,N'_{k-1}(z)) = c\#_1 z$, where $(N'_i(z),\dots,N'_{k-1}(z))$ denotes the output of N' corresponding to input z.*

Let $u > 0$ and $v \geq u$ be real numbers with $\lfloor v \rfloor - \lfloor u \rfloor \leq a$. Let $x = (x_1, \ldots, x_n)$ in $\{0, 1\}^n$ with $\#_1(x_1, \ldots, x_{i-1}) \leq u$ and $\#_1(x_1, \ldots, x_{k-1}) \leq v$. Let $y = (y_1, \ldots, y_n)$ denote the output of N corresponding to input x. Then

$$\#_1(y_1, y_2, \ldots, y_{j-1}) \leq u + c(v - u).$$

Proof. Note that condition 1 implies that

$$\#_1(x_1, x_2, \ldots, x_{i-1}) = \#_1(y_1, y_2, \ldots, y_{i-1}),$$
$$\#_1(x_i, x_{i+1}, \ldots, x_{k-1}) = \#_1(y_i, y_{i+1}, \ldots, y_{k-1}), \qquad \text{and}$$
$$\#_1(x_k, x_{k+1}, \ldots, x_n) = \#_1(y_k, y_{k+1}, \ldots, y_n).$$

Let $w = \#_1(x_i, x_{i+1}, \ldots, x_{k-1}) = \#_1(y_i, y_{i+1}, \ldots, y_{k-1})$, $\Delta u = u - \lfloor u \rfloor$, and $\Delta v = v - \lfloor v \rfloor$. Then, since $0 \leq c \leq 1$, we have $(1 - c)\Delta u + c\Delta v \geq 0$. Since $\lfloor u \rfloor + c(\lfloor v \rfloor - \lfloor u \rfloor) = u + c(v - u) - ((1 - c)\Delta u + c\Delta v) \leq u + c(v - u)$, it suffices to show that

$$\#_1(y_1, y_2, \ldots, y_{j-1}) \leq \lfloor u \rfloor + c(\lfloor v \rfloor - \lfloor u \rfloor).$$

There are two cases to consider.

Case 1. $w \geq \lfloor v \rfloor - \lfloor u \rfloor$.

Since $\#_1(x_1, x_2, \ldots, x_{k-1}) \leq v$ and $\#_1(x_1, x_2, \ldots, x_{k-1})$ is an integer,

$$\#_1(y_1, y_2, \ldots, y_{i-1}) = \#_1(x_1, x_2, \ldots, x_{k-1}) - w \leq \lfloor v \rfloor - w. \tag{1}$$

Since $w = \#_1(x_i, x_{i+1}, \ldots, x_{k-1}) \geq \lfloor v \rfloor - \lfloor u \rfloor$, we are able to take $x' = (x'_i, \ldots, x'_{k-1})$ such that $\#_1(x'_i, \ldots, x'_{k-1}) = \lfloor v \rfloor - \lfloor u \rfloor$ and $(x_i, \ldots, x_{k-1}) \geq (x'_i, \ldots, x'_{k-1})$. Let $(y'_i, \ldots, y'_{k-1})$ denote the output of N' corresponding to input $(x'_i, \ldots, x'_{k-1})$. Then by the fact that $a \geq \lfloor v \rfloor - \lfloor u \rfloor = \#_1(x'_i, \ldots, x'_{k-1})$, condition 2, and the monotonicity of N', we have

$$\#_1(y_j, \ldots, y_{k-1}) \geq \#_1(y'_j, \ldots, y'_{k-1}) \geq (1 - c)(\lfloor v \rfloor - \lfloor u \rfloor),$$

which implies

$$\#_1(y_i, y_{i+1}, \ldots, y_{j-1}) \leq w - (1 - c)(\lfloor v \rfloor - \lfloor u \rfloor). \tag{2}$$

Thus by expressions (1) and (2), we have

$$\begin{aligned}\#_1(y_1, y_2, \ldots, y_{j-1}) &\leq (\lfloor v \rfloor - w) + w - (1 - c)(\lfloor v \rfloor - \lfloor u \rfloor)\\ &= \lfloor u \rfloor + c(\lfloor v \rfloor - \lfloor u \rfloor).\end{aligned}$$

Case 2. $w < \lfloor v \rfloor - \lfloor u \rfloor$.

Since $\#_1(x_1, x_2, \ldots, x_{i-1}) \leq u$ and $\#_1(x_1, x_2, \ldots, x_{i-1})$ is an integer, we have

$$\#_1(y_1, y_2, \ldots, y_{i-1}) = \#_1(x_1, x_2, \ldots, x_{i-1}) \leq \lfloor u \rfloor. \tag{3}$$

On the other hand, by condition 2 and the inequality $a \geq \lfloor v \rfloor - \lfloor u \rfloor > w$, we have

$$\#_1(y_i, y_{i+1}, \ldots, y_{j-1}) \leq cw \leq c(\lfloor v \rfloor - \lfloor u \rfloor). \tag{4}$$

Thus by expressions (3) and (4), we have

$$\#_1(y_1, y_2, \ldots, y_{j-1}) \leq \lfloor u \rfloor + c(\lfloor v \rfloor - \lfloor u \rfloor). \qquad \square$$

The next lemma is the same as the previous one except that the condition bounding the number of 1's in an output vector is given in a different way from the condition in the previous lemma.

LEMMA 2.12. *Let N be a comparator network with n registers. Let i, j, and k be integers with $1 \leq i < j < k \leq n+1$. Let $a > 0$ and $c' \geq 0$ be some constants. Let a and c' be real constants with $a > 0$ and $c' \geq 0$. Assume that N satisfies the following conditions*:

1. *There is no comparator that connects a register in $\{1, 2, \ldots, i-1\}$ and a register in $\{i, i+1, \ldots, n\}$ and there is no comparator that connects a register in $\{1, 2, \ldots, k-1\}$ and a register in $\{k, k+1, \ldots, n\}$.*

2. *Let N' denote the comparator subnetwork of N composed of registers $i, i+1, \ldots, k-1$ and the associated comparators of N. For every z in $\{0,1\}^{k-i}$, $\#_1 z \leq a$ implies $\#_1(N'_i(z), \ldots, N'_{j-1}(z)) \leq c'$, where $(N'_i(z), \ldots, N'_{k-1}(z))$ denotes the output of N' corresponding to input z.*

Let $u > 0$ and $v \geq u$ be real numbers with $\lfloor v \rfloor - \lfloor u \rfloor \leq a$. Let $x = (x_1, \ldots, x_n)$ be a vector in $\{0,1\}^n$ with $\#_1(x_1, \ldots, x_{i-1}) \leq u$ and $\#_1(x_1, \ldots, x_{k-1}) \leq v$. Let $y = (y_1, \ldots, y_n)$ denote the output of N corresponding to input x. Then

$$\#_1(y_1, y_2, \ldots, y_{j-1}) \leq u + c'.$$

3. Construction of classifiers. In this section, we give the complete structure of our classifiers. Before describing the structure in detail, we need to give some definitions and notations.

For simplicity, the input size of our classifier, denoted n, is assumed to be even. We take arbitrarily real constants ε_C, θ_C, and δ_C and integer constant k_C satisfying

$$(5) \qquad 0 < \varepsilon_C, \quad 0 < \theta_C < 1, \quad 0 < \delta_C < 1 \quad \text{and} \quad k_C \geq \frac{2\left(1+\sqrt{1-\delta_C}\right)}{\delta_C} + \varepsilon_C.$$

Roughly speaking, our classifier is constructed, as is shown later in Fig. 3, by connecting $O(\log n)$ comparator networks, called layers, and removing unnecessary comparators from the layers.

As illustrated in Fig. 2, a typical layer is composed of compressors and an extractor put in the center of the layer. The submodule in the center of the layer put at the rightmost part is taken to be a sorter rather than an extractor.

In order to precisely specify these compressors, an extractor, and a sorter in a layer, we need to define sets of registers, which are called sections. Each section is denoted $X_n(i)$, $Y_n(i)$, or $Z_n(i)$. Based on sections defined in Definition 3.1, the modules in a layer, such as compressors, an extractor, and a sorter, will be defined in Definition 3.8.

DEFINITION 3.1. *For a positive integer i, $X_n(i)$ denotes the set $\{\lceil (n/2)(1-\theta_C^{i-1})\rceil + 1, \lceil (n/2)(1-\theta_C^{i-1})\rceil + 2, \ldots, \lceil (n/2)(1-\theta_C^{i})\rceil\}$ and $Y_n(i)$ denotes the set $\{n+1-x \mid x \in X_n(i)\}$. If $\lceil (n/2)(1-\theta_C^{i-1})\rceil = \lceil (n/2)(1-\theta_C^{i})\rceil$, then $X_n(i) = Y_n(i) = \emptyset$. For an integer $i \geq 0$, $Z_n(i)$ denotes the set $\{1, \ldots, n\} \setminus (\bigcup_{j=1}^{i} X_n(j) \cup \bigcup_{j=1}^{i} Y_n(j))$.*

We need to define some more constants which are determined by the constants ε_C, θ_C, δ_C, and k_C mentioned above. In the following lemmas, we state conditions concerning the constants. These conditions will be used later.

The compressors in layers are constructed using Lemma 2.9. Constants k_C and ε_C correspond to k and ε in the lemma, respectively. So $k_C n$ gives the upper bound on the size of the compressors except the outermost ones in a layer, whereas $k_C - 1 - \varepsilon_C$ represents the performance parameter of the same compressors. In the case of the outermost compressor in a layer, we take k_D (described in Definition 3.8) to correspond to parameter k in Lemma 2.9.

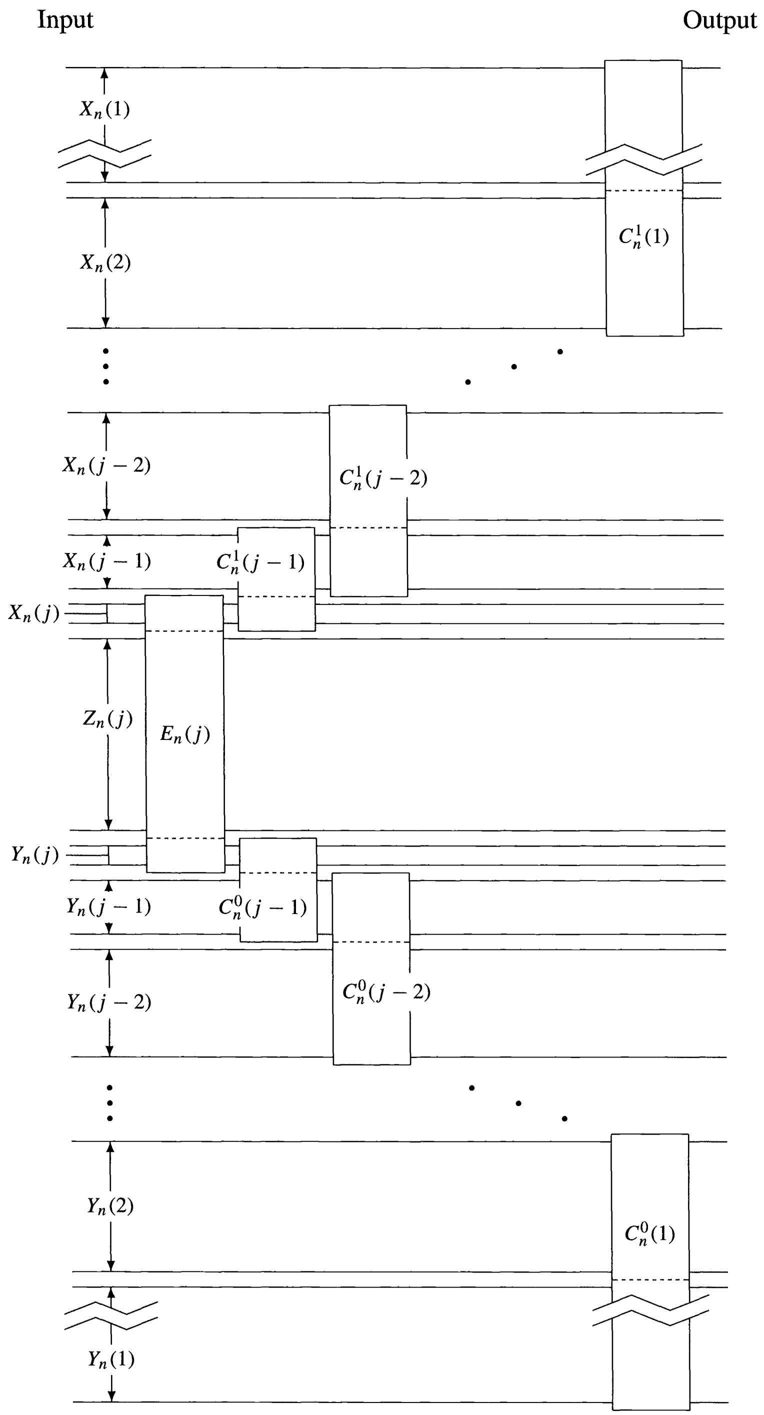


FIG. 2. *Structure of layer $L_n(j)$.*

So in the case of the outermost compressors, $k_D n$ and $k_D - 1 - \varepsilon_C$ give the upper bound on the size and the performance parameter, respectively.

DEFINITION 3.2. *Constants d_C, k_D, γ_C, Γ_0, Γ_1, and $j_{\max}(n)$ are given as*

$$d_C = \frac{1+\sqrt{1-\delta_C}}{\delta_C}, \qquad k_D = \max\left\{3 + \lfloor \varepsilon_C \rfloor, \left\lceil \frac{1}{\delta_C}\left(1 + \frac{2d_C}{k_C - \varepsilon_C}\right) + \varepsilon_C \right\rceil\right\},$$

$$\gamma_C = \frac{\theta_C \alpha_C}{2(k_C - \varepsilon_C)}, \qquad \Gamma_0 = \frac{\theta_C \alpha_C}{4}, \qquad \Gamma_1 = \frac{d_C \alpha_C}{2},$$

$$j_{\max}(n) = \max\left\{1, \left\lfloor \frac{\log_2 n - \log_2\left(\frac{2}{(1-\theta_C)^2}(\max\{2/\alpha_C, n_{\max}\} + 1)\right)}{\log_2(1/\theta_C)} \right\rfloor\right\},$$

where $n_{\max} = \max\{n_A(\theta_C/2, k_C, \varepsilon_C), n_A(\theta_C/2, k_D, \varepsilon_C), n_B(\gamma_C, (1-\theta_C)/(1+2\theta_C))\}$, *and* $\alpha_C = \min\{\alpha_A(\theta_C/2, k_C, \varepsilon_C), \alpha_A(\theta_C/2, k_D, \varepsilon_C)\}$.

LEMMA 3.3. *The following statements all hold*:

$$1 < d_C \delta_C < d_C, \tag{6}$$

$$\frac{2d_C}{k_C - \varepsilon_C} \le 1, \tag{7}$$

and

$$\frac{1}{d_C} + \sum_{i=1}^{j}\left(\frac{1}{d_C}\right)^i \le \delta_C \sum_{i=0}^{j}\left(\frac{1}{d_C}\right)^i \quad \textit{for any positive integer } j. \tag{8}$$

Proof. This lemma can be proved by straightforward calculation. □

LEMMA 3.4. *The following statements all hold*:

$$2 < k_D - \varepsilon_C, \tag{9}$$

$$k_D \le k_C, \tag{10}$$

and

$$\frac{1}{k_D - \varepsilon_C}\left(\frac{2d_C}{k_C - \varepsilon_C} + 1\right) \le \delta_C. \tag{11}$$

Proof. This lemma can be proved by straightforward calculation. □

The constant γ_C, corresponding to γ in Lemma 2.10, gives the performance parameter of an extractor in a layer, which is constructed by using the lemma.

LEMMA 3.5. *The following statement holds*:

$$\frac{k_C - \varepsilon_C}{2}\gamma_C = \Gamma_0. \tag{12}$$

Proof. This lemma can be proved by straightforward calculation. □

LEMMA 3.6. *The following statements all hold*:

$$(\theta_C/2)|X_n(j)| \le |X_n(j+1)| \quad \textit{for } j = 1, \ldots, j_{\max}(n) - 1, \tag{13}$$

$$|X_n(j+1)| \le |X_n(j)| \quad \textit{for } j = 1, \ldots, j_{\max}(n) - 1, \tag{14}$$

$$\frac{1-\theta_C}{1+2\theta_C}(|X_n(j)|+|Z_n(j)|) \le |X_n(j)| \quad \textit{for } j=1,\dots,j_{\max}(n), \tag{15}$$

$$|X_n(j) \cup Z_n(j) \cup Y_n(j)| \ge n_B(\gamma_C, (1-\theta_C)/(1+2\theta_C)) \ge 1 \quad \textit{for } j=1,\dots,j_{\max}(n), \tag{16}$$

$$|X_n(j)| \ge n_A(\theta_C/2, k_C, \varepsilon_C) \ge 1 \quad \textit{for } j=1,\dots,j_{\max}(n), \tag{17}$$

$$|X_n(j)| \ge n_A(\theta_C/2, k_D, \varepsilon_C) \ge 1 \quad \textit{for } j=1,\dots,j_{\max}(n), \tag{18}$$

and

$$|X_n(j)| \ge 2/\alpha_C > 2 \quad \textit{for } j=1,\dots,j_{\max}(n). \tag{19}$$

Proof. This lemma can be proved by direct calculation. □

Note that, because of ceilings appearing in Definition 3.1, the definition for $X_n(i)$ does not directly imply inequalities (13)–(15) in the previous lemma, especially when $|X_n(i)|$ is small.

Moreover, we can verify the following lemma by observing constraint (5), Definition 3.2, and the previous four lemmas.

LEMMA 3.7. *The following statements hold for the constants*:

$$\Gamma_0 < \Gamma_1, \tag{20}$$

$$d_C\Gamma_0|X_n(i)| \le \Gamma_1|X_n(i+1)| \quad \textit{for } i=1,\dots,j_{\max}(n)-1, \tag{21}$$

$$\gamma_C/\delta_C \le \Gamma_0, \tag{22}$$

$$\left(\frac{2}{k_C-\varepsilon_C}+\frac{1}{d_C}\right)\Gamma_1|X_n(j)| \le 2\alpha_C|X_n(j)|-2 \quad \textit{for } j=1,\dots,j_{\max}(n), \tag{23}$$

$$j_{\max}(n) = O(\log n), \tag{24}$$

and

$$|X_n(j_{\max}(n)) \cup Z_n(j_{\max}(n)) \cup Y_n(j_{\max}(n))| = |Z_n(j_{\max}(n)-1)| = O(1). \tag{25}$$

We are now ready to specify how to construct a layer, illustrated in Fig. 2, using modules such as compressors, an extractor, and a sorter.

DEFINITION 3.8. *$C_n^1(1)$ is an $(|X_n(1)|, |X_n(2)|, \alpha_A(\theta_C/2, k_D, \varepsilon_C), k_D-1-\varepsilon_C)$-compressor of type* 1 *that crosses the registers in $X_n(1) \cup X_n(2)$.*

$C_n^0(1)$ is a $(|Y_n(1)|, |Y_n(2)|, \alpha_A(\theta_C/2, k_D, \varepsilon_C), k_D-1-\varepsilon_C)$-compressor of type 0 *that crosses the registers in $Y_n(1) \cup Y_n(2)$.*

For $i=2,\dots,j_{\max}(n)-1$, $C_n^1(i)$ is an $(|X_n(i)|, |X_n(i+1)|, \alpha_A(\theta_C/2, k_C, \varepsilon_C), k_C-1-\varepsilon_C)$-compressor of type 1 *that crosses the registers in $X_n(i) \cup X_n(i+1)$.*

For $i=2,\dots,j_{\max}(n)-1$, $C_n^0(i)$ is a $(|Y_n(i)|, |Y_n(i+1)|, \alpha_A(\theta_C/2, k_C, \varepsilon_C), k_C-1-\varepsilon_C)$-compressor of type 0 *that crosses the registers in $Y_n(i) \cup Y_n(i+1)$.*

For $i=1,\dots,j_{\max}(n)-1$, $E_n(i)$ is a $(|Z_n(i)|, |X_n(i)|, \gamma_C)$-extractor that crosses the registers in $X_n(i) \cup Z_n(i) \cup Y_n(i)$.

$E_n(j_{\max}(n))$ is an $(|X_n(j_{\max}(n))| + |Z_n(j_{\max}(n))| + |Y_n(j_{\max}(n))|)$-sorter that crosses the registers in $X_n(j_{\max}(n)) \cup Z_n(j_{\max}(n)) \cup Y_n(j_{\max}(n))$.

As shown in inequalities (16)–(18), $j_{\max}(n)$ is taken to be small enough to make the sizes of $X_n(i)$, $Y_n(i)$, and $Z_n(i)$ large for $i = 1, \ldots, j_{\max}(n)$ so that $C_n^1(i)$ and $C_n^0(i)$ constructed according to Lemma 2.9 work as compressors for $i = 2, \ldots, j_{\max}(n)$ and $E_n(i)$ constructed according to Lemma 2.10 works as an extractor for $i = 1, \ldots, j_{\max}(n) - 1$. Moreover, as shown in (25), $j_{\max}(n)$ is also taken to be large enough to make the sizes of $X_n(j_{\max}(n))$, $Y_n(j_{\max}(n))$, and $Z_n(j_{\max}(n))$ less than some constant not depending on n.

LEMMA 3.9. *There exist $C_n^1(1), \ldots, C_n^1(j_{\max}(n)) - 1$, $C_n^0(1), \ldots, C_n^0(j_{\max}(n)) - 1$, and $E_n(1), \ldots, E_n(j_{\max}(n))$ such that the following hold:*

$$\text{size}\left(C_n^1(1)\right) \le k_D|X_n(1)|, \quad \text{size}\left(C_n^0(1)\right) \le k_D|X_n(1)|,$$

$$\text{depth}\left(C_n^1(1)\right) \le \lceil 2k_D/\theta_C \rceil, \quad \textit{and} \quad \text{depth}\left(C_n^0(1)\right) \le \lceil 2k_D/\theta_C \rceil;$$

for $j = 2, \ldots, j_{\max}(n) - 1$,

$$\text{size}\left(C_n^1(j)\right) \le k_C|X_n(j)|, \quad \text{size}\left(C_n^0(j)\right) \le k_C|X_n(j)|,$$

$$\text{depth}\left(C_n^1(j)\right) \le \lceil 2k_C/\theta_C \rceil, \quad \textit{and} \quad \text{depth}\left(C_n^0(j)\right) \le \lceil 2k_C/\theta_C \rceil;$$

for $j = 1, \ldots, j_{\max}(n)$,

$$\text{size}\left(E_n(j)\right) = O(|X_n(j)|) \quad \textit{and} \quad \text{depth}\left(E_n(j)\right) = O(1).$$

Proof. The lemma easily follows from Lemma 3.6, 2.9, and 2.10. □

For comparator networks M and N with the same number—say m—of registers, $M \circ N$ denotes the comparator network with m registers obtained by joining the ith output terminal of M to the ith input terminal of N for each $i = 1, 2, \ldots, m$.

DEFINITION 3.10. *Let n be an even positive integer. For $j = 1, 2, \ldots, j_{\max}(n)$, the* layer *of* rank *$j$ with n registers, denoted by $L_n(j)$, is defined as follows: $L_n(1) = E_n(1)$; for $j \ge 2$, $L_n(j) = E_n(j) \circ (C_n^1(j-1) \circ C_n^0(j-1)) \circ \cdots \circ (C_n^1(1) \circ C_n^0(1))$.*

The structure of layer $L_n(j)$ is illustrated in Fig. 2. We note here that there are generally many expressions describing the same comparator network. For example, layer $E_n(j) \circ (C_n^1(j-1) \circ C_n^0(j-1)) \circ \cdots \circ (C_n^1(1) \circ C_n^0(1))$ in the definition above can also be written as $E_n(j) \circ C_n^1(j-1) \circ \cdots \circ C_n^1(1) \circ C_n^0(j-1) \circ \cdots \circ C_n^0(1)$.

We are now ready to describe the comparator networks which we intend to be classifiers.

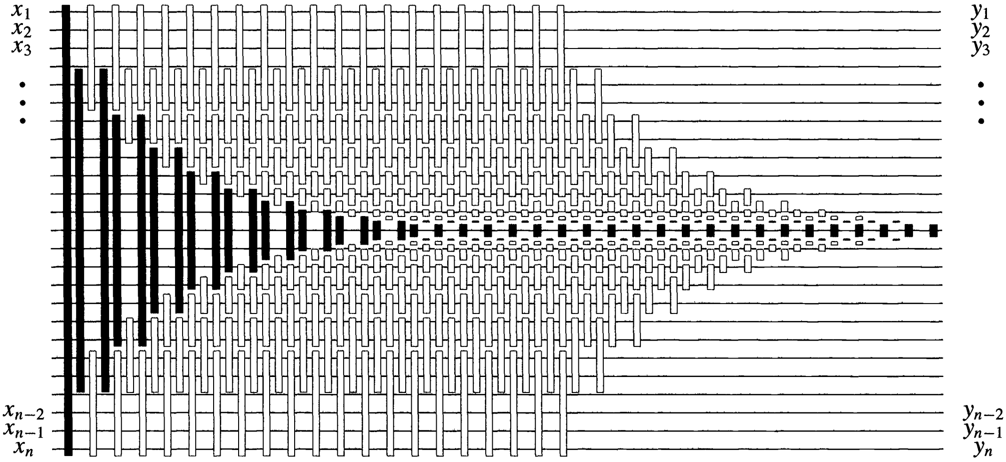
DEFINITION 3.11. *Given an even positive integer n, we define comparator network $\mathcal{N}^n$ with n input terminals as follows:*

1. *For a positive integer j and a nonnegative integer k, we define $g(j,k)$ as $d_C^{j-1}\delta_C^k g(1,0)$, where $g(1,0) = \Gamma_0|X_n(1)|$. In addition, $G(j,k)$ denotes $\sum_{i=1}^{j} g(i,k)$.*

2. *For a positive integer k, we define j_k as follows: if $j_{\max}(n) = 1$, then $j_k = j_{\max}(n) = 1$; else if $g(j_{\max}(n) - 1, k - 1) < \Gamma_0|X_n(j_{\max}(n) - 1)|$, then $j_k = j_{\max}(n)$; else $j_k = \min\{j \in \{1, \ldots, j_{\max}(n) - 1\} \mid g(j, k-1) \ge \Gamma_0|X_n(j)|\}$. In addition, we define j_0 to be 1. Let L_k denote $L_n(j_k)$.*

3. *For a nonnegative integer k, N_k denotes $L_0 \circ L_1 \circ \cdots \circ L_k$.*

4. *Let k be a nonnegative integer. We define $i_{\max}(k)$ as follows: if $G(1,k) \ge 1$, then $i_{\max}(k) = 0$; otherwise $i_{\max}(k) = \max\{i \in \{1, 2, \ldots, j_k\} \mid G(i,k) < 1\}$. Moreover, we define L'_k as follows: if $i_{\max}(k) \le 1$, then $L'_k = L_k$; otherwise $L'_k = E_n(j_k) \circ (C_n^1(j_k - 1) \circ C_n^0(j_k - 1)) \circ \cdots \circ (C_n^1(i_{\max}(k)) \circ C_n^0(i_{\max}(k)))$. Intuitively, L'_k is defined to be the comparator network obtained by removing $C_n^1(1), \ldots, C_n^1(i_{\max}(k) - 1)$, $C_n^0(1), \ldots$, and $C_n^0(i_{\max}(k) - 1)$ from L_k. We call L'_k as well as L_k a layer.*



Each framed rectangle represents a compressor, whereas each black rectangle represents an extractor or a sorter. The binary values on the registers pass from the left to the right.

FIG. 3. *A sketch of classifier $\mathcal{N}^n$.*

5. *For a nonnegative integer k, N'_k denotes $L'_0 \circ L'_1 \circ \cdots \circ L'_k$.*
6. *The expression $k_{\max}(n)$ denotes $\min\{k \in \mathbb{N} \mid j_k = j_{\max}(n)$ and $G(j_{\max}(n), k) < 1\}$.*
7. *The expression $\mathcal{N}^n$ denotes $N'_{k_{\max}(n)}$.*

Fig. 3 shows a sketch of $\mathcal{N}^n$.

As is illustrated in Fig. 3, the classifier given by Definition 3.11 is the comparator network with n input terminals on which labels $x_1, \ldots, x_n$ are put and n output terminals on which labels $y_1, \ldots, y_n$ are put. To construct the classifier, we first connect $k_{\max} + 1$ layers in cascade to obtain $L_0 \circ L_1 \circ \cdots \circ L_{k_{\max}}$ and then remove unnecessary compressors to obtain $L'_0 \circ L'_1 \circ \cdots \circ L'_{k_{\max}}$, which is a classifier, denoted by $\mathcal{N}^n$. Typically, layer L_k looks like the network given in Fig. 2 with j replaced by j_k. So layer L_k has $2(j_k - 1)$ compressors. By the definition of j_k, it is clear that $1 \le j_k \le j_{\max}(n)$. Moreover, by (6), (14), and the definition of function g, it will be shown that $j_k \le j_{k-1} + 1$ for any positive integer k (Lemma 4.2). So the number of compressors in a layer increases by at most 2 as the index k of layer L_k increases by 1. Integer $j_{\max}(n)$ gives the maximum integer among $j_1 \le \cdots \le j_{k_{\max}(n)}$, that is, $j_{k_{\max}(n)}$.

Before giving the proof that $\mathcal{N}^n$ defined above works as a classifier, we describe the rough idea behind the definition. Let $\mathbf{B}_n$ denote $\{x \in \{0,1\}^n \mid \#_0 x = \#_1 x (= n/2)\}$. By an argument similar to the proof of the zero-one principle for sorting networks in Knuth [5], we can derive that if a comparator network—say N—with n registers sorts any vector in $\mathbf{B}_n$, then N works as a classifier for any totally ordered input set. In what follows, we therefore confine input vectors to $\mathcal{N}^n$ to $\mathbf{B}_n$. Since a vector in $\mathbf{B}_n$ is assumed to be given as input to $\mathcal{N}^n$, it is natural to consider 1's appearing in registers $\{1, \ldots, n/2\}$ and 0's appearing in registers $\{n/2+1, \ldots, n\}$ at the output terminals of $\mathcal{N}^n$ to be error or impurity. For $S \subseteq \{1, \ldots, n\}$, let $((x_1, \ldots, x_n) \mid S)$ denote $(x_{s_1}, \ldots, x_{s_m})$, where $S = \{s_1, \ldots, s_m\}$ and $s_1 < \cdots < s_m$. The key parameter in the argument for verifying that $\mathcal{N}^n$ is a classifier is the error vector of a comparator network N which, as its component, gives the maximum number of the possible errors in each region of the output terminals of N, such as $\bigcup_{j=1}^{i} X_n(j)$ or $\bigcup_{j=1}^{i} Y_n(j)$ when N receives an arbitrary input vector in $\mathbf{B}_n$ as input.

DEFINITION 3.12. *Let N be a comparator network. For $i = 1, \ldots, j_{\max}(n)$, let* $\mathrm{Error}^i(N)$ *denote*

$$\max\left\{\max_{x\in\mathbf{B}_n}\#_1\left(N(x)\;\middle|\;\bigcup_{j=1}^{i}X_n(j)\right),\ \max_{x\in\mathbf{B}_n}\#_0\left(N(x)\;\middle|\;\bigcup_{j=1}^{i}Y_n(j)\right)\right\}.$$

Vector $(\mathrm{Error}^1(N), \ldots, \mathrm{Error}^{j_{\max}(n)}(N))$ *is called the* error vector *of N.*

It can be easily seen that since the module in the center of the rightmost layer of $\mathcal{N}^n$ is the sorter working on $X_n(j_{\max}(n)) \cup Z_n(j_{\max}(n)) \cup Y_n(j_{\max}(n))$, it follows that if there exists no impurity in $\bigcup_{j=1}^{j_{\max}(n)}(X_n(j) \cup Y_n(j))$ at the output terminals of $\mathcal{N}^n$, then there exists no impurity in any register at the output terminals, i.e., $\mathcal{N}^n$ works as a classifier for all of the input vectors in $\mathbf{B}_n$. So in order to verify that $\mathcal{N}^n$ works as a classifier, it suffices to show that the $j_{\max}(n)$th component of the error vector of $\mathcal{N}^n$ is 0.

For real-valued vectors $x = (x_1, \ldots, x_l)$ and $y = (y_1, \ldots, y_m)$, we write $x \preceq y$ if $l \geq m$ and $x_i \leq y_i$ for each $i = 1, \ldots, m$. It will be shown that for each $k = 1, \ldots, k_{\max}(n)$, $(G(1, k), G(2, k), \ldots, G(j_k, k))$ is an upper bound on the error vector of N_k $(= L_0 \circ L_1 \circ \cdots \circ L_k)$ under the relation $\preceq$ for vectors mentioned above, i.e., for each $k = 1, \ldots, k_{\max}(n)$ and each $i = 1, \ldots, j_k$, inequalities $\max_{x\in\mathbf{B}_n}\#_1(N_k(x) \mid \bigcup_{j=1}^{i}X_n(j)) \leq G(i, k)$ and $\max_{x\in\mathbf{B}_n}\#_0(N_k(x) \mid \bigcup_{j=1}^{i}Y_n(j)) \leq G(i, k)$ hold. So $G(i, k)$ can be regarded as the number of errors that we allow to appear in the regions $\bigcup_{j=1}^{i}X_n(j)$ or $\bigcup_{j=1}^{i}Y_n(j)$ at the output terminals of N_k. On the other hand, by the definition of $k_{\max}(n)$, we have $G(j_{k_{\max}(n)}, k_{\max}(n)) = G(j_{\max}(n), k_{\max}(n)) < 1$. Thus we can prove that the $j_{\max}(n)$th component of the error vector of $N_{k_{\max}(n)} (= \mathcal{N}^n)$ is zero.

The fact that $(G(1, k), G(2, k), \ldots, G(j_k, k))$ is an upper bound on the error vector of $N_k = L_0 \circ L_1 \circ \cdots \circ L_k$ for $k = 1, \ldots, k_{\max}(n)$ will be proved by induction on k. Roughly speaking, this fact will be verified by showing that whenever a layer is connected to the right side of the comparator network, the performance of the network is improved by the factor of $\delta_C < 1$ with respect to the error vector. To make the statement precise, we need to introduce condition A and property B described in Definition 3.13 below. Setting $j = j_k$, let $G_1 = G(1, k-1), \ldots, G_j = G(j, k-1)$. As the induction hypothesis, we assume that $(G_1, \ldots, G_{j-1})$ is an upper bound on the error vector of N_{k-1}. Furthermore, we can see that $(G_1, \ldots, G_j)$ satisfies condition A in Definition 3.13. Then we can show that $\delta_c(G_1, \ldots, G_j) = (G(1, k), \ldots, G(j, k))$ is an upper bound on the error vector of $N_{k-1} \circ L_n(j)$ (Lemma 4.8). Thus it is concluded that $L_n(j)$ has the property of improving the error vector by the factor of δ_C. We call this property B$[j]$. Once it is proved that $(G(1, k), \ldots, G(j_k, k))$ is an upper bound on the error vector of N_k for $k = 1, \ldots k_{\max}(n)$, it can be shown that N_k behaves exactly the same way as N'_k for $k = 1, \ldots, k_{\max}(n)$ (Lemma 4.9). This is because N'_k is obtained from N_k by removing unnecessary compressors. Recall that compressor $C_n^1(i)$ in layer L_k, which works on $X_n(i) \cup X_n(i+1)$, is unnecessary if $G(i+1, k) < 1$ holds, i.e., there exists no impurity in $X_n(1) \cup \cdots \cup X_n(i+1)$ at the output terminal of L_k, and similarly for $C_n^0(i)$.

The precise definitions for condition A and property B$[j]$ are given as follows.

DEFINITION 3.13. *Let $(G_1, \ldots, G_j)$ be a vector whose components are nonnegative real numbers. Vector $(G_1, \ldots, G_j)$ is said to satisfy condition* A *if the following three conditions are satisfied*:

A1. *If $j \geq 2$, then $G_2 - G_1 = d_C G_1$ and, for any $i = 2, \ldots, j-1$, $G_{i+1} - G_i = d_C(G_i - G_{i-1})$.*

A2. *$G_1 \leq \Gamma_1|X_n(1)|$ when $j = 1$, and $G_j - G_{j-1} \leq \Gamma_1|X_n(j)|$ otherwise.*

A3. *$j = j_{\max}(n)$, or if $j = 1$, then $G_1 \geq \Gamma_0|X_n(1)|$; otherwise $G_j - G_{j-1} \geq \Gamma_0|X_n(j)|$.*

Let j be a positive integer not greater than $j_{\max}(n)$. Comparator network L is said to have property B$[j]$ *if the following conditions are satisfied: If $(G_1, \dots, G_j)$ satisfies condition* A *and if $j = 1$ or $(G_1, \dots, G_{j-1})$ is an upper bound on the error vector of N, then $\delta_C(G_1, \dots, G_j)$ is an upper bound on the error vector of $N \circ L$.*

Before closing this section, we discuss briefly the size and the depth of $\mathcal{N}^n$. To estimate the depth of our classifier $\mathcal{N}^n$, we first notice that any module in $\mathcal{N}^n$, such as a compressor, an extractor, or a sorter, has constant depth. Then we verify that any path from an input terminal to an output terminal in $\mathcal{N}^n$ passes at most $O(k_{\max}(n) + j_{\max}(n))$ modules. Finally, by showing that $j_{\max}(n) = O(\log n)$ and $k_{\max}(n) = O(\log n)$ (Lemma 4.12 in the next section), we conclude that the depth of $\mathcal{N}^n$ is $O(\log n)$.

To estimate the size of $\mathcal{N}^n$ we notice that the same module may appear repeatedly in different layers. For example, compressor $C_n^1(i)$ working on $X_n(i) \cup X_n(i+1)$ may appear in several layers. It turns out that the size of the compressors dominates the size of $\mathcal{N}^n$. We shall show that any kind of compressor appears in at most $\log_2 n / \log_2(1/\delta_C)$ distinct layers in $\mathcal{N}^n$. So, by setting constants ε_C, θ_C, δ_C, and k_C suitably, we can show that the total size of our classifier is $Cn \log_2 n + O(n)$ for any constant C greater than $3/\log_2 3$ (Theorem 4.14 in the next section).

4. Analysis of comparator network $\mathcal{N}^n$. In this section, we shall analyze the behavior of comparator network $\mathcal{N}^n$ and give a strict proof of the fact that $\mathcal{N}^n$ is a classifier for every even positive integer n. Then we shall evaluate the size and depth of the classifier, proving the main result (Theorem 4.14).

LEMMA 4.1. *For any positive integers j and k, condition $g(j, k-1) \geq \Gamma_0 |X_n(j)|$ is equivalent to condition $j_k \leq j$.*

Proof. It follows from the definition of function g and fact (14) that $g(x, k-1) - \Gamma_0|X_n(x)|$ is strictly monotone increasing with respect to x on the set of positive integers. Thus the lemma follows. □

LEMMA 4.2.

(26) *For any nonnegative integer k,* $1 \leq j_k \leq j_{\max}(n)$,

and

(27) *for any positive integer k,* $j_k \leq j_{k-1} + 1$.

Proof. Statement (26) is obvious by the definition of j_k. We shall show that statement (27) holds.

First, assume that $k = 1$. Then we have $g(1, k-1) = g(1, 0) = \Gamma_0|X_n(1)|$. By Lemma 4.1, we have $j_k \leq 1$, which verifies that $j_k \leq 2 = j_{k-1} + 1$. Next, assume that $k \geq 2$. By the definition of j_{k-1}, we have $g(j_{k-1}, k-2) \geq \Gamma_0|X_n(j_{k-1})|$. By inequalities (6) in Lemma 3.3 and (14) in Lemma 3.6, we have $g(j_{k-1} + 1, k-1) = d_C \delta_C g(j_{k-1}, k-2) > g(j_{k-1}, k-2) \geq \Gamma_0|X_n(j_{k-1})| \geq \Gamma_0|X_n(j_{k-1} + 1)|$. Thus by Lemma 4.1, we have $j_k \leq j_{k-1} + 1$. This completes the proof of the lemma. □

LEMMA 4.3. *For any integer k with $1 \leq k \leq k_{\max}(n)$, $(G(1, k-1), G(2, k-1), \dots, G(j_k, k-1))$ satisfies condition* A.

Proof. Let k be an integer in $\{1, \dots, k_{\max}(n)\}$. By the definitions of G and j_k, it is obvious that $(G(1, k-1), G(2, k-1), \dots, G(j_k, k-1))$ satisfies conditions A1 and A3. We shall show that $(G(1, k-1), G(2, k-1), \dots, G(j_k, k-1))$ also satisfies condition A2, that is,

$$g(j_k, k-1) \leq \Gamma_1 |X_n(j_k)|.$$

First, assume that $j_k = 1$. Then, since $\Gamma_0 < \Gamma_1$ and $g(1, k-1) = \delta_C^{k-1} g(1, 0) \leq g(1, 0)$ by definition, we have $g(j_k, k-1) = g(1, k-1) \leq g(1, 0) = \Gamma_0|X_n(1)| < \Gamma_1|X_n(1)| =$

$\Gamma_1|X_n(j_k)|$. Next, assume that $j_k > 1$. Then, by the definition of j_k, we have $d_C g(j_k - 1, k-1) < d_C\Gamma_0|X_n(j_k-1)|$. Moreover, $g(j_k, k-1) = d_C g(j_k - 1, k-1)$ follows from the definition of function g, $d_C\Gamma_0|X_n(j_k-1)| \le (2d_C/\theta_C)\Gamma_0|X_n(j_k)|$ follows from fact (13) in Lemma 3.6, and, finally, $(2d_C/\theta_C)\Gamma_0|X_n(j_k)| = \Gamma_1|X_n(j_k)|$ follows from the definitions of Γ_0 and Γ_1. Thus we have $g(j_k, k-1) < \Gamma_1|X_n(j_k)|$, completing the proof of the lemma. □

In the following lemmas, N denotes a comparator network, typically a subnetwork consisting of modules in the left part of $\mathcal{N}^n$.

LEMMA 4.4. *The inequality*

$$\mathrm{Error}^1(E_n(1)) \le \gamma_C|X_n(1)|$$

holds. Moreover, for any $a \ge 0$ *and any integer* j *with* $2 \le j \le j_{\max}(n)$, *if* $\mathrm{Error}^{j-1}(N) \le a$, *then*

$$\mathrm{Error}^j(N \circ E_n(j)) \le a + \gamma_C|X_n(j)|$$

holds.

Proof. By the definition of $E_n(1)$ and the definition of extractors, the inequality

$$\mathrm{Error}^1(E_n(1)) \le \gamma_C|X_n(1)|$$

immediately follows.

Assume that $\mathrm{Error}^{j-1}(N) \le a$. We shall show that

$$\mathrm{Error}^j(N \circ E_n(j)) \le a + \gamma_C|X_n(j)|$$

holds. Let x be a vector in $\mathbf{B}_n$. Since $\mathrm{Error}^{j-1}(N) \le a$, we have

$$\#_1(N(x) \mid X_n(1) \cup \cdots \cup X_n(j-1)) \le a$$

and

$$\#_0(N(x) \mid Y_n(1) \cup \cdots \cup Y_n(j-1)) \le a.$$

It is easy to see that the latter implies

$$\#_1(N(x) \mid X_n(1) \cup \cdots \cup X_n(j) \cup Z_n(j) \cup Y_n(j)) \le \frac{1}{2}|Z_n(j-1)| + a,$$

whereas the former implies

$$\#_0(N(x) \mid Y_n(1) \cup \cdots \cup Y_n(j) \cup Z_n(j) \cup X_n(j)) \le \frac{1}{2}|Z_n(j-1)| + a.$$

Moreover, since $(1/2)|Z_n(j-1)|$ is an integer, we have

$$\left\lfloor \frac{1}{2}|Z_n(j-1)| + a \right\rfloor - \lfloor a \rfloor = \frac{1}{2}|Z_n(j-1)| = \frac{1}{2}|Z_n(j)| + |X_n(j)| \le |Z_n(j)| + |X_n(j)|.$$

If $j = j_{\max}(n)$, then $E_n(j)$ is a sorter. Applying Lemma 2.12 to the sorter, we have

$$\#_1(E_n(j)(N(x)) \mid X_n(1) \cup \cdots \cup X_n(j)) \le a \le a + \gamma_C|X_n(j)|$$

and

$$\#_0(E_n(j)(N(x)) \mid Y_n(1) \cup \cdots \cup Y_n(j)) \le a \le a + \gamma_C|X_n(j)|,$$

which imply $\mathrm{Error}^j(N \circ E_n(j)) \le a + \gamma_C|X_n(j)|$. Otherwise, $E_n(j)$ is a $(|Z_n(j)|, |X_n(j)|, \gamma_C)$-extractor. Applying Lemma 2.12 to the extractor, we have

$$\#_1(E_n(j)(N(x)) \mid X_n(1) \cup \cdots \cup X_n(j)) \le a + \gamma_C|X_n(j)|$$

and

$$\#_0(E_n(j)(N(x)) \mid Y_n(1) \cup \cdots \cup Y_n(j)) \le a + \gamma_C|X_n(j)|,$$

which imply $\mathrm{Error}^j(N \circ E_n(j)) \le a + \gamma_C|X_n(j)|$, completing the proof of the lemma. □

The following two lemmas easily follow from Lemma 2.11 and the definition of a compressor.

LEMMA 4.5. *For any $b \ge 0$, any $c \ge 0$, and any integer j with $2 \le j \le j_{\max}(n) - 1$, if $\mathrm{Error}^{j-1}(N) \le b$, $\mathrm{Error}^{j+1}(N) \le c$, and $c - b \le \lceil (k_C - \varepsilon_C)\lfloor \alpha_C|X_n(j)|\rfloor\rceil$, then*

$$\mathrm{Error}^j(N \circ C_n^1(j) \circ C_n^0(j)) \le b + \frac{c-b}{k_C - \varepsilon_C}$$

holds.

LEMMA 4.6. *For any $d \ge 0$, if $\mathrm{Error}^2(N) \le d \le \lceil (k_D - \varepsilon_C)\lfloor \alpha_C|X_n(1)|\rfloor\rceil$, then*

$$\mathrm{Error}^1(N \circ C_n^1(1) \circ C_n^0(1)) \le \frac{d}{k_D - \varepsilon_C}$$

holds.

LEMMA 4.7. *For each $j = 1, \ldots, j_{\max}(n)$, $L_n(j)$ has property* B[j].

Proof. It follows from Lemma 4.4, (22), and condition A3 that $L_n(1)$ has property B[1]. In the rest of this proof, assume that $j \ge 2$. Let $G = (G_1, \ldots, G_j)$ be a vector of nonnegative real numbers that satisfies condition A. Assume that $(G_1, \ldots, G_{j-1})$ is an upper bound on the error vector of N. Let M_j denote $E_n(j)$. Moreover, for an integer i with $1 \le i \le j - 1$, let M_i denote the subnetwork of $L_n(j)$ consisting of $C_n^1(i), \ldots, C_n^1(j-1), C_n^0(i), \ldots, C_n^0(j-1)$ and $E_n(j)$, that is, $M_i = E_n(j) \circ (C_n^1(j-1) \circ C_n^0(j-1)) \circ \cdots \circ (C_n^1(i) \circ C_n^0(i))$. Notice that M_i exactly crosses registers in $X_n(i) \cup Z_n(i) \cup Y_n(i)$.

Now, by induction on i, we shall show that

$$\mathrm{Error}^i(N \circ M_i) \le \frac{2}{k_C - \varepsilon_C}(G_i - G_{i-1}) + G_{i-1} \tag{28}$$

holds for each $i = 2, \ldots, j$. First, we show that inequality (28) holds for $i = j$, which is the induction basis.

By the assumption, applying Lemma 4.4 to $M_j = E_n(j)$ yields

$$\mathrm{Error}^j(N \circ M_j) \le \gamma_C|X_n(j)| + G_{j-1}.$$

Moreover, by condition A3, we have $G_j - G_{j-1} \ge \Gamma_0|X_n(j)|$. Therefore, it follows from inequality (12) that

$$\begin{aligned}\mathrm{Error}^j(N \circ M_j) &\le \gamma_C|X_n(j)| + G_{j-1}\\ &= \frac{2}{k_C - \varepsilon_C}\Gamma_0|X_n(j)| + G_{j-1}\\ &\le \frac{2}{k_C - \varepsilon_C}(G_j - G_{j-1}) + G_{j-1}.\end{aligned}$$

Thus, expression (28) holds for $i = j \ge 2$.

Next, let r be an integer with $2 < r \leq j$ and assume that expression (28) holds for $i = r$. By inequality (6), condition A2, facts (23) and (14), and the fact that $k_C - \varepsilon_C > 2$, which follows from the constraint $k_C \geq \left(2\left(1 + \sqrt{1 - \delta_C}\right)/\delta_C\right) + \varepsilon_C$ in (5), we have

$$\begin{aligned}
&\left\lfloor \frac{2}{k_C - \varepsilon_C}(G_r - G_{r-1}) + G_{r-1} \right\rfloor - \lfloor G_{r-2} \rfloor \\
&\qquad = \left\lfloor \frac{2}{k_C - \varepsilon_C}(G_r - G_{r-1}) + G_{r-1} - \lfloor G_{r-2} \rfloor \right\rfloor \\
&\qquad \leq \left\lfloor \frac{2}{k_C - \varepsilon_C}(G_r - G_{r-1}) + (G_{r-1} - G_{r-2}) + 1 \right\rfloor \\
&\qquad = \left\lfloor \left(\frac{2}{k_C - \varepsilon_C} + \frac{1}{d_C}\right)(G_r - G_{r-1}) + 1 \right\rfloor \\
&\qquad \leq \left\lfloor \left(\frac{2}{k_C - \varepsilon_C} + \frac{1}{d_C}\right)(G_j - G_{j-1}) + 1 \right\rfloor \\
&\qquad \leq \left\lfloor \left(\frac{2}{k_C - \varepsilon_C} + \frac{1}{d_C}\right)\Gamma_1 |X_n(j)| + 1 \right\rfloor \\
&\qquad \leq \lfloor 2\alpha_C |X_n(j)| - 1 \rfloor \\
&\qquad \leq \lfloor \alpha_C |X_n(j)| + \lfloor \alpha_C |X_n(j)| \rfloor \rfloor \\
&\qquad = 2 \lfloor \alpha_C |X_n(j)| \rfloor \\
&\qquad \leq 2 \lfloor \alpha_C |X_n(r-1)| \rfloor \\
&\qquad \leq \lceil (k_C - \varepsilon_C) \lfloor \alpha_C |X_n(r-1)| \rfloor \rceil .
\end{aligned} \tag{29}$$

On the other hand, by the induction hypothesis, we have

$$\mathrm{Error}^r(N \circ M_r) \leq \frac{2}{k_C - \varepsilon_C}(G_r - G_{r-1}) + G_{r-1}.$$

Moreover, since M_r does not cross any register in $X_n(1) \cup \cdots \cup X_n(r-2)$ and $\mathrm{Error}^i(N) \leq G_i$ for each $i = 1, \ldots, j-1$, we have

$$\mathrm{Error}^{r-2}(N \circ M_r) = \mathrm{Error}^{r-2}(N) \leq G_{r-2}.$$

Since inequality (29) holds, we have, by Lemma 4.5,

$$\begin{aligned}
&\mathrm{Error}^{r-1}(N \circ M_r \circ C_n^1(r-1) \circ C_n^0(r-1)) \\
&\qquad = \mathrm{Error}^{r-1}(N \circ M_{r-1}) \leq \frac{\frac{2}{k_C - \varepsilon_C}(G_r - G_{r-1}) + G_{r-1} - G_{r-2}}{k_C - \varepsilon_C} + G_{r-2}.
\end{aligned}$$

Hence it follows from inequality (7) that

$$\begin{aligned}
\mathrm{Error}^{r-1}(N \circ M_{r-1}) &\leq \frac{\frac{2}{k_C - \varepsilon_C}(G_r - G_{r-1}) + G_{r-1} - G_{r-2}}{k_C - \varepsilon_C} + G_{r-2} \\
&= \frac{\frac{2d_C}{k_C - \varepsilon_C}(G_{r-1} - G_{r-2}) + (G_{r-1} - G_{r-2})}{k_C - \varepsilon_C} + G_{r-2} \\
&\leq \frac{2}{k_C - \varepsilon_C}(G_{r-1} - G_{r-2}) + G_{r-2}.
\end{aligned}$$

Therefore, expression (28) holds for $i = r - 1$. Thus we have shown that expression (28) holds for each $i = 2, \ldots, j$. Moreover, by inequality (7) and fact (8), the following inequality holds for each $i = 2, \ldots, j$. In the rest of this proof, let G_0 denote 0.

$$\begin{aligned}
\mathrm{Error}^i(N \circ M_i) &\le \frac{2}{k_C - \varepsilon_C}(G_i - G_{i-1}) + G_{i-1} \\
&= \frac{2}{k_C - \varepsilon_C}(G_i - G_{i-1}) + \sum_{j=1}^{i-1}(G_j - G_{j-1}) \\
&= \frac{2}{k_C - \varepsilon_C}(G_i - G_{i-1}) + (G_i - G_{i-1})\sum_{h=1}^{i-1}\left(\frac{1}{d_C}\right)^h \\
&= (G_i - G_{i-1})\left(\frac{2}{k_C - \varepsilon_C} + \sum_{h=1}^{i-1}\left(\frac{1}{d_C}\right)^h\right) \\
&\le (G_i - G_{i-1})\left(\frac{1}{d_C} + \sum_{h=1}^{i-1}\left(\frac{1}{d_C}\right)^h\right) \\
&\le \delta_C(G_i - G_{i-1})\sum_{h=0}^{i-1}\left(\frac{1}{d_C}\right)^h \\
&= \delta_C\sum_{j=1}^{i}(G_j - G_{j-1}) = \delta_C G_i.
\end{aligned}$$

Now we shall show that $\mathrm{Error}^1(N \circ M_1) = \mathrm{Error}^1(N \circ L_n(j)) \le \delta_C G_1$. Since expression (28) holds for $i = 2$ and $G_2 - G_1 = d_C G_1$, we have

$$\mathrm{Error}^2(N \circ M_2) \le \frac{2}{k_C - \varepsilon_C}(G_2 - G_1) + G_1 = \left(\frac{2d_C}{k_C - \varepsilon_C} + 1\right)G_1. \tag{30}$$

Moreover, by condition A1, condition A2, and fact (14), we have $G_2 - G_1 \le G_j - G_{j-1} \le \Gamma_1|X_n(j)| \le \Gamma_1|X_n(2)|$. By fact (23) and inequality (9), we have

$$\begin{aligned}
\mathrm{Error}^2(N \circ M_2) &\le \frac{2}{k_C - \varepsilon_C}(G_2 - G_1) + G_1 \\
&= \left(\frac{2}{k_C - \varepsilon_C} + \frac{1}{d_C}\right)(G_2 - G_1) \\
&\le \left(\frac{2}{k_C - \varepsilon_C} + \frac{1}{d_C}\right)\Gamma_1|X_n(2)| \\
&\le 2\alpha_C|X(2)| - 2 \le 2\alpha_C|X(1)| - 2 \\
&\le 2\lfloor\alpha_C|X(1)|\rfloor \\
&\le \lceil(k_D - \varepsilon_C)\lfloor\alpha_C|X(1)|\rfloor\rceil.
\end{aligned} \tag{31}$$

By (30), (31), Lemma 4.6, and (11), we have

$$\begin{aligned}
\mathrm{Error}^1(N \circ M_2 \circ C_n^1(1) \circ C_n^0(1)) &= \mathrm{Error}^1(N \circ M_1) \\
&\le \frac{1}{k_D - \varepsilon_C}\left(\frac{2d_C}{k_C - \varepsilon_C} + 1\right)G_1 \\
&\le \delta_C G_1.
\end{aligned}$$

Thus we have shown that $\mathrm{Error}^i(N \circ L_n(j)) \leq \mathrm{Error}^i(N \circ M_i) \leq \delta_C G_i$ holds for each $i = 1, \ldots, j$, that is, $\delta_C(G_1, \ldots, G_j)$ is an upper bound on the error vector of $N \circ L_n(j)$. Note that any component of the error vector does not increase after attaching a comparator network. □

LEMMA 4.8. *For each $k = 0, \ldots, k_{\max}(n)$, $(G(1,k), \ldots, G(j_k, k))$ is an upper bound on the error vector of N_k.*

Proof. We shall prove the lemma by induction on k.

First, we shall prove the basis. By inequality (22), we have $\gamma_C < \Gamma_0$. Since $N_0 = L_n(1) = E_n(1)$ is a $(|Z_n(1)|, |X_n(1)|, \gamma_C)$-extractor or a sorting network, we have $\#_1(N_0(x) \mid X_n(1)) \leq \gamma_C|X_n(1)| < \Gamma_0|X_n(1)|$ and $\#_0(N_0(x) \mid Y_n(1)) \leq \gamma_C|X_n(1)| < \Gamma_0|X_n(1)|$ for any x in $\mathbf{B}_n$. Therefore, $(G(1,0)) = (G(j_0, 0))$ is an upper bound on the error vector of N_0.

Next, we shall prove the induction step. Let k be a positive integer. Assume that $(G(1, k-1), \ldots, G(j_{k-1}, k-1))$ is an upper bound on the error vector of N_{k-1}. It follows from fact (27) in Lemma 4.2 that $(G(1, k-1), \ldots, G(j_k - 1, k-1))$ is an upper bound on the error vector of N_{k-1}. Since Lemma 4.3 implies that $(G(1, k-1), \ldots, G(j_k, k-1))$ satisfies condition A and Lemma 4.7 implies that $L_k = L_n(j_k)$ has property B$[j]$, it follows that $\delta_C(G(1, k-1), \ldots, G(j_k, k-1)) = (G(1,k), \ldots, G(j_k, k))$ is an upper bound on the error vector of $N_{k-1} \circ L_k = N_k$, completing the proof of the lemma. □

LEMMA 4.9. *For any integer k with $1 \leq k \leq k_{\max}(n)$ and any x in $\mathbf{B}_n$, $N_k(x) = N'_k(x)$.*

Proof. The proof is by induction on k. By definition, it is clear that $N_0 = N'_0$. Let k be a positive integer less than or equal to $k_{\max}(n)$, and let x be an arbitrary vector in $\mathbf{B}_n$. Assume that $N_{k-1}(x) = N'_{k-1}(x)$. If $L_k = L'_k$, then we have $N_k(x) = L_k(N_{k-1}(x)) = L'_k(N'_{k-1}(x)) = N'_k(x)$. Assume that $L_k \neq L'_k$, and hence $i_{\max}(n) \geq 2$. By the definition of $i_{\max}(k)$, any compressor in L_k but not in L'_k does not intersect any register in $Z_n(i_{\max}(k))$. Moreover, since $(G(1,k), G(2,k), \ldots, G(j_k, k))$ is an upper bound on the error vector of N_k, all of the impurities in the output terminals of N_k appear in $Z_n(i_{\max}(k))$ by the definition of $i_{\max}(k)$. We therefore have

$$(32)\qquad \begin{aligned}(L_k(N_{k-1}(x)) \mid Z_n(i_{\max}(k))) &= (L_k(N'_{k-1}(x)) \mid Z_n(i_{\max}(k)))\\ &= (L'_k(N'_{k-1}(x)) \mid Z_n(i_{\max}(k))),\end{aligned}$$

$$(33)\qquad \begin{aligned}\left(L_k(N_{k-1}(x)) \;\middle|\; \bigcup_{i=1}^{i_{\max}(k)} X_n(i)\right) &= \left(L_k(N'_{k-1}(x)) \;\middle|\; \bigcup_{i=1}^{i_{\max}(k)} X_n(i)\right)\\ &= \left(L'_k(N'_{k-1}(x)) \;\middle|\; \bigcup_{i=1}^{i_{\max}(k)} X_n(i)\right)\\ &= (0, 0, \ldots, 0),\end{aligned}$$

and

$$(34)\qquad \begin{aligned}\left(L_k(N_{k-1}(x)) \;\middle|\; \bigcup_{i=1}^{i_{\max}(k)} Y_n(i)\right) &= \left(L_k(N'_{k-1}(x)) \;\middle|\; \bigcup_{i=1}^{i_{\max}(k)} Y_n(i)\right)\\ &= \left(L'_k(N'_{k-1}(x)) \;\middle|\; \bigcup_{i=1}^{i_{\max}(k)} Y_n(i)\right)\\ &= (1, 1, \ldots, 1).\end{aligned}$$

By equations (32)–(34), we have $N_k(x) = (N_{k-1} \circ L_k)(x) = L_k(N_{k-1}(x)) = L'_k(N'_{k-1}(x)) = (N'_{k-1} \circ L'_k)(x) = N'_k(x)$. This completes the proof of the lemma. □

LEMMA 4.10. *For any even positive integer n, $\mathcal{N}^n$ is a classifier.*

Proof. Let x be an arbitrary vector in $\mathbf{B}_n$. It follows from Lemmas 4.8 and 4.9 that we obtain $\#_1(\mathcal{N}^n(x) \mid \bigcup_{i=1}^{j_{\max}(n)} X_n(i)) = 0$ and $\#_0(\mathcal{N}^n(x) \mid \bigcup_{i=1}^{j_{\max}(n)} Y_n(i)) = 0$, and hence $\#_1 (\mathcal{N}^n(x) \mid Z_n(j_{\max}(n))) = \#_0 (\mathcal{N}^n(x) \mid Z_n(j_{\max}(n)))$. Moreover, since the module $E_n(j_{\max}(n))$ at the center of the rightmost layer is a sorting network, $(\mathcal{N}^n(x) \mid Z_n(j_{\max}(n)))$ is sorted. We therefore conclude that vector $\mathcal{N}^n(x)$ is sorted, completing the proof of the lemma. □

LEMMA 4.11. $k_{\max}(n) = O(\log n)$.

Proof. Since $0 < g(1, 0) = \Gamma_0|X_n(1)| = (\theta_C\alpha_C/4)|X_n(1)| < n/2$ and $j_{\max}(n) = O(\log n)$, there exist constants $s > 0$ and $t > 0$ such that

$$G(j_{\max}(n), 0) = g(1, 0) \sum_{i=1}^{j_{\max}(n)} d_C^{i-1} < sn^t.$$

Let r denote $\lceil \log_{1/\delta_C}(sn^t / \min\{1, d_C\delta_C\Gamma_0|X_n(j_{\max}(n) - 1)|\})\rceil$, and hence $\delta_C^r \leq 1/sn^t$ and $\delta_C^r \leq d_C\delta_C\Gamma_0|X_n(j_{\max}(n) - 1)|/sn^t$. Since $\Gamma_0|X_n(j_{\max}(n) - 1)| = O(1)$, we have $r = O(\log n)$. Moreover, we have

$$G(j_{\max}(n), r) = \delta_C^r G(j_{\max}(n), 0) < \frac{1}{sn^t} sn^t = 1$$

and

$$\begin{aligned} G(j_{\max}(n) - 1, r - 1) &= \delta_C^{r-1} \sum_{i=1}^{j_{\max}(n)-1} g(i, 0) = d_C^{-1}\delta_C^{r-1} \sum_{i=2}^{j_{\max}(n)} g(i, 0) \\ &< \frac{\delta_C^r}{d_C\delta_C} G(j_{\max}(n), 0) \leq \frac{1}{d_C\delta_C} \frac{d_C\delta_C\Gamma_0|X_n(j_{\max}(n) - 1)|}{sn^t} sn^t \\ &= \Gamma_0|X_n(j_{\max}(n) - 1)|. \end{aligned}$$

By the definition of $k_{\max}(n)$, we have $k_{\max}(n) \leq r = O(\log n)$, completing the proof of the lemma. □

LEMMA 4.12. depth$(\mathcal{N}^n) = O(\log n)$.

Proof. By definition, the modules in $\mathcal{N}^n$ are clearly of constant depth. Furthermore, by (25) in Lemma 3.7, all of the sorters in $\mathcal{N}^n$ are of constant depth. So to estimate the depth of $\mathcal{N}^n$, it suffices to count the maximum number of modules, i.e., compressors, extractors, and sorters, that appear on a path from an input terminal of $\mathcal{N}^n$ to an output terminal. To do so, we assign an integer to each module in $\mathcal{N}^n$ as the level. For a module in the kth layer $L_n(j_k)$, an integer is assigned as the level as follows: $E_n(j_k)$ is assigned $2k$ and both $C_n^1(j)$ and $C_n^0(j)$ are assigned $2k + j_k - j$, where $k = 0, 1, \ldots, k_{\max}$ and $j = 1, 2, \ldots, j_k - 1$. Since $j_k \leq j_{\max}(n)$ for $k = 0, \ldots, k_{\max}(n)$, the level of any module in $\mathcal{N}^n$ is not greater than $2k_{\max}(n) + j_{\max}(n) - 1$. On the other hand, it can be easily seen that if a module of level l is connected to a module of level l', then $l' \geq l + 1$. So if a path from an input terminal to an output terminal in $N_{k_{\max}(n)}$ crosses m modules, then

$$m \leq 2k_{\max}(n) + j_{\max}(n).$$

Since $j_{\max}(n) = O(\log n)$ by Lemma 24, and $k_{\max}(n) = O(\log n)$ by Lemma 4.11, we have $m = O(\log n)$, completing the lemma. □

LEMMA 4.13. *For any positive real numbers ε_C, $\theta_C < 1$, and $\delta_C < 1$ and any integer k_C with $k_C \geq \frac{2(1+\sqrt{1-\delta_C})}{\delta_C} + \varepsilon_C$,*

$$\mathrm{size}(\mathcal{N}^n) \leq F(\varepsilon_C, \theta_C, \delta_C, k_C) n \log_2 n + O(n),$$

where

$$F(\varepsilon_C, \theta_C, \delta_C, k_C) = \frac{(1-\theta_C)k_D + \theta_C k_C}{\log_2(1/\delta_C)}.$$

Proof. Let $s_1(k,n)$ denote the total size of the compressors in L'_k. Let $s_2(k,n)$ and $s_3(k,n)$ be defined as follows: $s_2(k,n) = \mathrm{size}(E_n(j_k))$ if $j_k \leq j_{\max}(n) - 1$, and $s_2(k,n) = 0$ otherwise; $s_3(k,n) = \mathrm{size}(E_n(j_k))$ if $j_k = j_{\max}(n)$, and $s_3(k,n) = 0$ otherwise. Note that if $j \leq j_{\max}(n) - 1$, then $E_n(j)$ is some extractor and $E_n(j_{\max}(n))$ is some sorting network. Clearly,

$$\mathrm{size}(\mathcal{N}^n) = \sum_{k=0}^{k_{\max}(n)} (s_1(k,n) + s_2(k,n) + s_3(k,n)) \tag{35}$$

holds. Since $\max_{0 \leq k \leq k_{\max}(n)} s_3(k,n) = O(1)$ follows from equation (25) in Lemma 3.7 and $k_{\max}(n) = O(\log n)$ holds by Lemma 4.11, we have

$$\sum_{k=0}^{k_{\max}(n)} s_3(k,n) = O(\log n). \tag{36}$$

Let h denote the constant $\lceil \log_2(2d_C/\theta_C)/\log_2(1/\delta_C) \rceil$, so that $\delta_C^h(2d_C/\theta_C) \leq 1$. We shall show that

$$\text{for } k = 1, \ldots, j_{\max}(n) - 1, \quad j_{k+h} \geq j_k + 1 \tag{37}$$

holds. Let k be a positive integer less than $j_{\max}(n)$. By the definitions of j_k and function g and by fact (13) in Lemma 3.6, we have $g(j_k, k-1) = d_C g(j_k - 1, k-1) < d_C \Gamma_0 |X_n(j_k - 1)| \leq (2d_C/\theta_C)\Gamma_0 |X_n(j_k)|$ for $j_k \geq 2$. On the other hand, by the definitions of the constant d_C and function g and by constraint (5), we also have $g(1, k-1) = \delta_C^{k-1} g(1,0) \leq \Gamma_0 |X_n(1)| < (2d_C/\theta_C)\Gamma_0 |X_n(1)|$. We therefore have $g(j_k, k-1) < (2d_C/\theta_C)\Gamma_0 |X_n(j_k)|$ for any k. By the definition of h, we obtain $g(j_k, k+h-1) = \delta_C^h g(j_k, k-1) < \delta_C^h (2d_C/\theta_C)\Gamma_0 |X_n(j_k)| \leq \Gamma_0 |X_n(j_k)|$. Thus we conclude by Lemma 4.1 that statement (37) holds.

It follows from (37) that for each nonnegative integer j less than $j_{\max}(n)$, the number of positive integers k with $j_k = j$ is less than or equal to $h+1$. Note that $j_k = j_{k+1} = \cdots = j_{k+h}$ occurs only when $k = 0$. Since $E_n(j)$ is contained in $L_n(i)$ exactly when $i = j$, and since there exists a constant C such that $\mathrm{size}(E_n(j)) \leq C|X_n(j)|$ for $j = 1, \ldots, j_{\max}(n) - 1$, we have

$$\sum_{k=0}^{k_{\max}(n)} s_2(k,n) \leq \sum_{j=1}^{j_{\max}(n)-1} C(h+1)|X_n(j)| = O(n). \tag{38}$$

Let j be a positive integer less than $j_{\max}(n)$. Assume that L'_k contains both $C_n^1(j)$ and $C_n^0(j)$, and hence $k \geq 1$. It is obvious that $j+1 \leq j_k$, and hence $G(j+1,k) \leq G(j_k,k)$. Since $(G(1,k-1), G(2,k-1), \ldots, G(j_k,k-1))$ satisfies condition A2 because of Lemma 4.3, we have $g(j_k, k-1) \leq \Gamma_1 |X_n(j_k)|$. It therefore follows that for any positive integer i less than or equal to j_k, $g(i,k-1) \leq g(j_k,k-1) \leq \Gamma_1 |X_n(j_k)| \leq \Gamma_1 |X_n(i)|$. We therefore have $G(j+1,k) = \delta_C \sum_{i=1}^{j+1} g(i,k-1) \leq \delta_C \Gamma_1 \sum_{i=1}^{j+1} |X_n(i)| \leq (\delta_C \Gamma_1/2)n = (\delta_C d_C \alpha_C/4)n = ((1+\sqrt{1-\delta_C})/4)\alpha_C n < n$. On the other hand, since if $G(j+1,k) < 1$ then L'_k contains neither $C_n^1(j)$ nor $C_n^0(j)$ by item 4 in Definition 3.11, we have $G(j+1,k) \geq 1$. Thus we have $1 \leq G(j+1,k) < n$. Let k_1 and k_2 be nonnegative integers. Since $G(j+1,k+1) = \delta_C G(j+1,k)$ for every nonnegative integer k, if $G(j+1,k_1) < n$ and $k_2 \geq k_1 + \log_{1/\delta_C} n$, then $G(j+1,k_2) < 1$. It therefore follows that the number of nonnegative integers k such that $1 \leq G(j+1,k) < n$ is less than $1 + \log_{1/\delta_C} n = 1 + (\log_2 n / \log_2(1/\delta_C))$. The number of nonnegative integers k such that L'_k contains both $C_n^1(j)$ and $C_n^0(j)$ is therefore less than $1 + (\log_2 n / \log_2(1/\delta_C))$ for each $j = 1, 2, \ldots, j_{\max}(n) - 1$.

Since $\operatorname{size}(C_n^1(1)) = \operatorname{size}(C_n^0(1)) \leq k_D|X_n(1)|$ and $\operatorname{size}(C_n^1(j)) = \operatorname{size}(C_n^0(j)) \leq k_C|X_n(j)|$ for each $j = 2, \ldots, j_{\max}(n) - 1$ by Lemma 3.9, we have

$$\sum_{k=0}^{k_{\max}(n)} s_1(k,n)$$
$$\leq \left(1 + \frac{\log_2 n}{\log_2(1/\delta_C)}\right)(k_D(|X_n(1)| + |Y_n(1)|) + k_C(n - |X_n(1)| - |Y_n(1)|)).$$

Since $k_D \leq k_C$ and $|X_n(1)| = |Y_n(1)| \geq (n/2)(1-\theta_C)$, we have

$$\sum_{k=0}^{k_{\max}(n)} s_1(k,n) \leq \frac{n \log_2 n}{\log_2(1/\delta_C)}((1-\theta_C)k_D + \theta_C k_C) + k_C n. \tag{39}$$

Thus the lemma follows from expressions (35), (36), (38) and (39). □

THEOREM 4.14. *For every $C > 3/\log_2 3 = 1.8927\ldots$, there exists a family of classifiers with an even number of input terminals such that its depth is $O(\log n)$ and its size is at most $Cn\log_2 n + O(n)$, where n is the number of input terminals.*

Proof. Take the constant ε_C so that $0 < \varepsilon_C < 1$. Take θ_C, δ_C, and k_C as

$$\delta_C = \frac{1}{3 - 2\varepsilon_C},$$

$$k_C = \left\lceil \frac{2d_C}{\varepsilon_C\delta_C} + \varepsilon_C \right\rceil = \left\lceil \frac{2\left(1+\sqrt{1-\delta_C}\right)}{\varepsilon_C\delta_C^2} + \varepsilon_C \right\rceil \geq \left\lceil \frac{2\left(1+\sqrt{1-\delta_C}\right)}{\delta_C} + \varepsilon_C \right\rceil,$$

and

$$\theta_C = \frac{\varepsilon_C}{k_C}$$

for constant ε_C. Condition (5) clearly holds. Then we have $3 + \lfloor \varepsilon_C \rfloor = 3$. Moreover, since $k_C - \varepsilon_C \geq 2d_C/\varepsilon_C\delta_C$, we have

$$\left\lceil \frac{1}{\delta_C}\left(1 + \frac{2d_C}{k_C - \varepsilon_C}\right) + \varepsilon_C \right\rceil \leq \left\lceil \frac{1 + \varepsilon_C\delta_C}{\delta_C} + \varepsilon_C \right\rceil$$
$$= \left\lceil \frac{1}{\delta_C} + 2\varepsilon_C \right\rceil = \lceil 3 - 2\varepsilon_C + 2\varepsilon_C \rceil = 3.$$

By the definition of k_D, we have $k_D = 3$. It follows from $\log_2(1/\delta_C) = \log_2(3 - 2\varepsilon_C)$, $\theta_C k_C = \varepsilon_C$, and $k_D = 3$ that

$$F(\varepsilon_C, \theta_C, \delta_C, k_C) \leq \frac{3 + \varepsilon_C}{\log_2(3 - 2\varepsilon_C)}.$$

Since $\frac{3+\varepsilon_C}{\log_2(3-2\varepsilon_C)} \to \frac{3}{\log_2 3}$ $(\varepsilon_C \to 0)$, we can take $\varepsilon_C > 0$ so that $F(\varepsilon_C, \theta_C, \delta_C, k_C) < C$. □

Finally, we note that no matter how we choose the constants ε_C, θ_C, δ_C, and k_C, the constant factor $3/\log_2 3$ in the upper bound of Theorem 4.14 cannot be improved as long as we follow the argument in the present paper. To verify the statement, it suffices to show that $k_D/\log_2(1/\delta_C) \geq 3/\log_2 3$ holds for any ε_C, θ_C, δ_C, and k_C because $F(\varepsilon_C, \theta_C, \delta_C, k_C) = ((1-\theta_C)k_D + \theta_C k_C)/\log_2(1/\delta_C) \geq k_D/\log_2(1/\delta_C)$. By $k_D \geq 1/\delta_C$, $k_D \geq 3$, and the fact that $x/\log_2 x$ takes the minimum value $3/\log_2 3$ in the range of x in which x is an integer greater than or equal to 3, we have $k_D/\log_2(1/\delta_C) \geq 3/\log_2 3$. The fact that $k_D \geq 1/\delta_C$ and $k_D \geq 3$ is obtained immediately from the definition of k_D.

5. Concluding remarks. In this paper, we investigate a method of constructing a classifier in a relatively simple way and obtain the result that, for any constant C greater than $3/\log_2 3$, there exists a family of classifiers with n inputs and n outputs such that the size and depth of the classifiers are bounded from above by $Cn\log_2 n + O(n)$ and $O(\log n)$, respectively.

We have tried to obtain as good a constant in the size complexity as possible at the cost of the constant in the depth complexity. In order to obtain an upper bound on the depth in the form $c_1 \log_2 n + o(\log n)$ with some coefficient c_1 explicitly given, we would seem to have to do a much more complicated analysis in the proof of Proposition 2.6 in the appendix. As to making the size complexity more precise, we could possibly obtain the upper bound of the size complexity in the form $Cn\log_2 n + c_2 n + o(n)$ by analyzing in more detail the size complexity of the extractors used in the classifiers. We note that both of the coefficients, c_1 and c_2, grow to infinity as C tends to $3/\log_2 3$.

If, generalizing the notion of a classifier, one wishes to classify n values into the t smallest values and the $n-t$ largest values, this can be accomplished by modifying the definition of an extractor, as shown in the following definition, and imitating the construction of classifiers given in the present paper, provided that $t = \Omega(n)$. All of the ratios m_1 to m_2 of the extractors used in the generalized classifier are taken to be nearly equal to the ratio t to $n-t$.

DEFINITION. *Let n, m_1, and m_2 be positive integers and γ be a real number with $0 \leq \gamma \leq 1$. An (n, m_1, m_2, γ)-*extractor *is a comparator network N with $n+m_1+m_2$ registers such that for every $x = (x_1, \ldots, x_{n+m_1+m_2})$ in $\{0,1\}^{n+m_1+m_2}$, $\#_1 x \leq n + m_2$ implies $\#_1(y_1, \ldots, y_{m_1}) \leq \gamma m_1$ and $\#_0 x \leq n+m_1$ implies $\#_0(y_{n+m_1+1}, \ldots, y_{n+m_1+m_2}) \leq \gamma m_2$, where $(y_1, \ldots, y_{n+m_1+m_2})$ is the output of N corresponding to x.*

The upper bound on the size of a family of the generalized classifiers constructed in this way is given by $C'n\log_2 n + O(n)$ for some $C' > 3/\log_2 3$. This situation, in which we cannot make C' smaller than or equal to $3/\log_2 3$, is the same as the case where $t = n/2$. In other words, as long as we imitate the construction of classifiers given in the present paper, the coefficient of $n\log_2 n$ in any upper bounds on the size of the generalized classifiers remains the same no matter what value t takes. This is mainly because the total size of extractors in a comparator network is $O(n)$ and the shape of a compressor does not depend on t.

Alekseev [3] gave lower bounds of $(n-t)\lceil\log_2(t+1)\rceil$ on the size of comparator networks which classify n values into the t smallest values and the $n-t$ largest values. Those lower bounds are estimated to be $n\log_2 n - O(n \log\log n)$ when t takes the value $n/\log_2 n$. So at present there remains a gap between the coefficients in the lower bounds and those in the upper bounds for the size of classifiers.

Appendix: Proof of Proposition 2.6. In the beginning of the proof, as we mentioned previously, we mainly follow Bassalygo's example [4]. The adjacency matrix of a bipartite graph with n left vertices and m right vertices is defined as an $n \times m$ matrix whose (i, j)-component is equal to the number of edges directed from the ith left vertex to the jth right vertex. It is not hard to see that an (α, β)-expander with n left vertices and m right vertices corresponds to a matrix that does not contain any $k \times (m - \lceil\beta k\rceil + 1)$ null submatrix, where $k = 1, 2, \ldots, \lfloor\alpha n\rfloor$.

Let E_{pqn} denote the set of $pqn \times pqn$ matrices of 0's and 1's, containing exactly one 1 in each column and in each row. The number of such matrices is $(pqn)!$. We consider ordered sums of s such matrices. We take the order of choice of the terms of the sums into consideration. Notice that the s matrices are not necessary different from each other. The number of such sums is $((pqn)!)^s$. Let B be that sum, whose value is a matrix of dimension $pqn \times pqn$, and let b_{hl} denote the (h, l)-component of the value of B. Let $M(B)$ denote the $qn \times pn$ matrix whose (i, j)-component, denoted by m_{ij}, is given by

$$\sum_{\substack{h \equiv i \pmod{qn} \\ l \equiv j \pmod{pn}}} b_{hl}.$$

Sum B will be called *poor* if $M(B)$ contains a $k \times (pn - \lceil \beta k \rceil + 1)$ null submatrix for some integer k with $1 \le k \le \lfloor \alpha qn \rfloor$. Since the number of matrices in E_{pqn} that have a fixed $pk \times q(pn - \lceil \beta k \rceil + 1)$ null submatrix is equal to

$$\begin{aligned} &\binom{pqn - pk}{q(pn - \lceil \beta k \rceil + 1)} \cdot (q(pn - \lceil \beta k \rceil + 1))! \cdot (pqn - q(pn - \lceil \beta k \rceil + 1))! \\ &\qquad = \frac{(pqn - pk)!\,(q\lceil \beta k \rceil - q)!}{(q\lceil \beta k \rceil - pk - q)!}, \end{aligned}$$

the number of poor sums with a fixed $k \times (pn - \lceil \beta k \rceil + 1)$ null submatrix in the corresponding $qn \times pn$ matrices is given by

$$\left(\frac{(pqn - pk)!\,(q\lceil \beta k \rceil - q)!}{(q\lceil \beta k \rceil - pk - q)!} \right)^s.$$

The number of different $k \times (pn - \lceil \beta k \rceil + 1)$ submatrices of a $qn \times pn$ matrix is $\binom{qn}{k}\binom{pn}{\lceil \beta k \rceil - 1}$. Consequently, the number of poor sums does not exceed

$$\sum_{k=1}^{\lfloor \alpha qn \rfloor} \binom{qn}{k} \binom{pn}{\lceil \beta k \rceil - 1} \left(\frac{(pqn - pk)!\,(q\lceil \beta k \rceil - q)!}{(q\lceil \beta k \rceil - pk - q)!} \right)^s.$$

The statement that this quantity is less than the number of all sums is equivalent to the inequality

$$(40) \qquad \sum_{k=1}^{\lfloor \alpha qn \rfloor} \binom{qn}{k} \binom{pn}{\lceil \beta k \rceil - 1} \left(\binom{q\lceil \beta k \rceil - q}{pk} \Big/ \binom{pqn}{pk} \right)^s < 1.$$

So if this inequality holds, then there exists a *good*, i.e., not poor, sum which, as we can readily see, corresponds to the adjacency matrix of an (α, β)-expander with qn inputs and pn outputs. The number of edges of this expander is $spqn$, where s can be taken to be any integer for which inequality (40) is satisfied. The rest of the proof will prove inequality (40).

Let $A(\beta, p, q, n, s, k)$ denote

$$\binom{qn}{k} \binom{pn}{\lceil \beta k \rceil - 1} \left(\binom{q\lceil \beta k \rceil - q}{pk} \Big/ \binom{pqn}{pk} \right)^s.$$

Fixing k and s, $A(\beta, p, q, n, s, k)$ can be represented by a rational expression $g(n)/f(n)$ such that $f(n)$ is a polynomial of degree spk, $g(n)$ is a polynomial of degree $k + \lceil \beta k \rceil - 1$, and the highest-degree terms of $f(n)$ and $g(n)$ are both positive. The following inequalities are assumptions of the proposition:

$$(41) \qquad 0 < \alpha < \frac{p}{\beta q} < 1,$$

$$s > \frac{H(\alpha) + (p/q)H(\alpha\beta q/p)}{pH(\alpha) - \alpha\beta q H(p/\beta q)}, \tag{42}$$

and

$$s > \frac{p(1+\beta) - \alpha\beta(p+q)}{p(p-\alpha\beta q)}. \tag{43}$$

It follows from inequalities (41) and (43) that

$$\begin{aligned} s &> \frac{p(1+\beta) - \alpha\beta(p+q)}{p(p-\alpha\beta q)} = \frac{p + p\beta - p\alpha\beta - q\alpha\beta}{p(p-\alpha\beta q)} \\ &> \frac{p + p\beta - q\alpha\beta^2 - q\alpha\beta}{p(p-\alpha\beta q)} = \frac{1+\beta}{p}. \end{aligned}$$

Hence we have

$$s > \frac{1+\beta}{p}. \tag{44}$$

We therefore have $spk > k + \beta k > k + \lceil \beta k \rceil - 1$, which implies $g(n)/f(n) \to +0 \quad (n \to +\infty)$. Thus for any $\varepsilon > 0$ and any integer $k \geq 1$, we can take an integer $n_1(\beta, p, q, s, \varepsilon, k) > 0$ such that

(45) for any integer $n \geq n_1(\beta, p, q, s, \varepsilon, k), \quad A(\beta, p, q, n, s, k) < \varepsilon.$

Let $k_0(\beta, p, q, s)$ denote $\max\{\lfloor 2q/(\beta q - p) \rfloor + 1, \lceil 2/(sp - 1 - \beta) \rceil\}$. Notice that since $sp - 1 - \beta > 0$ follows from inequality (44) and $\beta q - p > 0$ follows from inequality (41), k_0 is always defined. Let us define

$$A_1(\beta, p, q, n, s) = \max_{k=1,2,\ldots,k_0-1} A(\beta, p, q, n, s, k)$$

and

$$A_2(\alpha, \beta, p, q, n, s) = \begin{cases} \displaystyle\max_{k=k_0,k_0+1,\ldots,\lfloor \alpha q n \rfloor} A(\beta, p, q, n, s, k) & \text{if } \alpha q n \geq k_0, \\ 0 & \text{otherwise.} \end{cases}$$

Then we have

$$\sum_{k=1}^{\lfloor \alpha q n \rfloor} A(\beta, p, q, n, s, k) \leq k_0 A_1(\beta, p, q, n, s) + \alpha q n A_2(\alpha, \beta, p, q, n, s). \tag{46}$$

Letting $n_2(\beta, p, q, s) = \max\limits_{1 \leq k < k_0} n_1(\beta, p, q, s, 1/2k_0, k)$, it follows from (45) that

(47) for every integer $n \geq n_2(\beta, p, q, s), \quad k_0 A_1(\beta, p, q, n, s) < \dfrac{1}{2}.$

Moreover, we shall find two integers n_5 and n_7 depending on α, β, p, q, and s such that,

(48) for any integer n with $n \geq n_5(\alpha, \beta, p, q, s)$ and k with $k_0 \leq k \leq \min\{\alpha q n, pn/2\beta\}$,

$$A(\beta, p, q, n, s, k) < \frac{1}{2\alpha q n}$$

and

(49) for any integer n with $n \geq n_7(\alpha, \beta, p, q, s)$ and k with $pn/2\beta < k \leq \alpha qn$,
$$A(\beta, p, q, n, s, k) < \frac{1}{2\alpha qn}.$$

In the following two paragraphs, we shall show that if we find n_5 and n_7, then we can conclude the proposition.

Since $k_0 \leq k \leq \min\{\alpha qn, pn/2\beta\}$ or $pn/2\beta < k \leq \alpha qn$ holds for any integer k with $k_0 \leq k \leq \alpha qn$,

for any integer n with $n \geq \max\{n_5, n_7\}$ and k with $k_0 \leq k \leq \alpha qn$,
$$A(\beta, p, q, n, s, k) < \frac{1}{2\alpha qn};$$

hence

(50) for any integer n with $n \geq \max\{n_5, n_7\}$, $\alpha qnA_2(\alpha, \beta, p, q, n, s) < \frac{1}{2}$.

Consequently, letting
$$s_0 = \left\lfloor \max\left\{\frac{H(\alpha) + (p/q)H(\alpha\beta q/p)}{pH(\alpha) - \alpha\beta qH(p/\beta q)}, \frac{p(1+\beta) - \alpha\beta(p+q)}{p(p-\alpha\beta q)}\right\}\right\rfloor + 1$$

and
$$n_0(\alpha, \beta, p, q) = \max\{n_2(\beta, p, q, s_0), n_5(\alpha, \beta, p, q, s_0), n_7(\alpha, \beta, p, q, s_0)\},$$

by facts (46), (47), and (50), we have the fact that
$$\text{for every integer } n \geq n_0(\alpha, \beta, p, q), \quad \sum_{k=1}^{\lfloor \alpha qn \rfloor} A(\beta, p, q, n, s_0, k) < 1.$$

Let n and k be integers such that $n \geq n_0(\alpha, \beta, p, q)$ and $1 \leq k \leq \alpha qn$. Since $q\lceil \beta k \rceil - q < pqn$ follows from inequality (41), we have
$$\binom{q\lceil \beta k \rceil - q}{pk} \Big/ \binom{pqn}{pk} < 1.$$

Thus, since $A(\beta, p, q, n, s, k)$ is monotone decreasing with respect to s, we conclude that $\sum_{k=1}^{\lfloor \alpha qn \rfloor} A(\beta, p, q, n, s, k) < 1$—i.e., inequality (40)—holds for any integer $s \geq s_0$ and $n \geq n_0(\alpha, \beta, p, q)$ provided that there exist n_5 and n_7 satisfying conditions (48) and (49).

Now, let k be a positive integer. We assume that $k_0 \leq k \leq \alpha qn$ in order to find n_5 and n_7. It is not hard to see that the following four inequalities hold:

(51) $$1 \leq k \leq \lfloor \alpha qn \rfloor \leq qn - 1,$$

(52) $$1 \leq \lceil \beta k \rceil - 1 \leq pn - 1,$$

(53) $$1 \leq pk \leq p\lfloor \alpha qn \rfloor \leq pqn - 1, \quad \text{and}$$

$$1 \le pk \le q\lceil \beta k\rceil - q - 1. \tag{54}$$

Inequality (54) is derived from the definition of k_0. We have $k \ge k_0 > 2q/(\beta q - p) > 0$ by definition. We therefore have $q\beta k > pk + 2q \ge pk + q + 1$, and hence $pk \le q\lceil \beta k\rceil - q - 1$. Moreover, we have the following fact by Stirling's formula:

(55) for every integer $n \ge 2$ and k with $1 \le k \le n-1$,

$$\begin{aligned}\frac{1}{3}\sqrt{\frac{n}{k(n-k)}}\exp\left(nH\left(\frac{k}{n}\right)\right) < \binom{n}{k} &< \frac{1}{2}\sqrt{\frac{n}{k(n-k)}}\exp\left(nH\left(\frac{k}{n}\right)\right)\\ &< \exp\left(nH\left(\frac{k}{n}\right)\right).\end{aligned}$$

We therefore have

$$\begin{aligned}A(\beta, p, q, n, s, k) < \left(\frac{3}{2}\right)^{s}\left(\frac{(q\lceil\beta k\rceil - q)(pqn - pk)}{(q\lceil\beta k\rceil - q - pk)\,pqn}\right)^{s/2}\exp\Bigg(&qnH\left(\frac{k}{qn}\right)\\ &+ pnH\left(\frac{\lceil\beta k\rceil - 1}{pn}\right)\\ &+ s\left(q\left(\lceil\beta k\rceil - 1\right)H\left(\frac{pk}{q\left(\lceil\beta k\rceil - 1\right)}\right) - pqnH\left(\frac{pk}{pqn}\right)\right)\Bigg).\end{aligned} \tag{56}$$

Notice that k/qn, $(\lceil\beta k\rceil - 1)/pn$ and $pk/q(\lceil\beta k\rceil - 1)$ are all greater than 0 and less than 1. That follows from inequalities (51), (52), and (54). On the other hand, since

$$\frac{d}{dz}\frac{H(z)}{z} = \frac{\log(1-z)}{z^2} < 0$$

holds for any z with $0 < z < 1$,

(57) $H(z)/z$ is strictly monotone decreasing over the interval $0 < z \le 1$.

We therefore have

$$q\left(\lceil\beta k\rceil - 1\right)H\left(\frac{pk}{q\left(\lceil\beta k\rceil - 1\right)}\right) < \beta qkH\left(\frac{p}{\beta q}\right). \tag{58}$$

Since $q/k \le q/k_0$ holds by the assumption and $q/k_0 \le (\beta q - p)/2$ holds by the definition of k_0, we have

$$\begin{aligned}\frac{(q\lceil\beta k\rceil - q)(pqn - pk)}{(q\lceil\beta k\rceil - q - pk)\,pqn} &\le \frac{\beta qk - q}{\beta qk - q - pk} = \frac{\beta q - (q/k)}{(\beta q - p) - (q/k)}\\ &\le \frac{\beta q - (q/k_0)}{(\beta q - p) - (q/k_0)} \le \frac{\beta q - (\beta q - p)/2}{(\beta q - p) - (\beta q - p)/2} = \frac{\beta q + p}{\beta q - p}.\end{aligned} \tag{59}$$

By inequalities (56), (58), and (59), we have

$$\begin{aligned}A(\beta, p, q, n, s, k) < \left(\frac{3}{2}\right)^{s}\left(\frac{\beta q + p}{\beta q - p}\right)^{s/2}\exp\Bigg(&qnH\left(\frac{k}{qn}\right) + pnH\left(\frac{\lceil\beta k\rceil - 1}{pn}\right)\\ &+ s\left(\beta qkH\left(\frac{p}{\beta q}\right) - pqnH\left(\frac{k}{qn}\right)\right)\Bigg).\end{aligned} \tag{60}$$

In what follows, x, C_1, C_2, and F will denote

$$x = \frac{k}{qn},$$

$$C_1(\beta, p, q, s) = \log\left(\left(\frac{3}{2}\right)^s \left(\frac{\beta q + p}{\beta q - p}\right)^{s/2}\right),$$

$$C_2(\beta, p, q, s) = s\beta q^2 H\left(\frac{p}{\beta q}\right), \qquad \text{and}$$

$$F(z) = q(1 - sp)H(z) + pH(\beta qz/p) + C_2(\beta, p, q, s)z,$$

respectively. The domain of $F(z)$ is the interval $0 \le z \le \alpha$. It follows from the equation $\frac{d^2}{dz^2}H(z) = \frac{-1}{z(1-z)}$ that

$$\frac{d^2}{dz^2}F(z) = \frac{q\left(p^2 s - p(1+\beta) - (pqs - p - q)\beta z\right)}{z(1-z)(p - \beta qz)}.$$

We have $sp > 1+\beta$ and $p < \beta q$ by inequalities (44) and (41), respectively. These inequalities imply $spq - p - q > (1+\beta)q - \beta q - q = 0$. Moreover, the inequalities $p^2 s - p(1+\beta) - (pqs - p - q)\beta\alpha > 0$ and $p - \beta q\alpha > 0$ follow from inequalities (43) and (41), respectively. The inequality $\frac{d^2}{dz^2}F(z) > 0$ therefore holds for any z with $0 < z \le \alpha$. Hence we have the fact that

(61) for any closed interval $[a, b] \subseteq [0, \alpha]$ and any z with $a < z < b$,
$$F(z) < \max\{F(a), F(b)\}.$$

It follows from fact (61), $F(0) = 0$, and $F(\alpha) < 0$ that

(62) $$F(z) < 0 \quad \text{for any } z \text{ with } 0 < z \le \alpha.$$

The inequality $F(\alpha) = q(1 - sp)H(\alpha) + pH(\beta q\alpha/p) + s\alpha\beta q^2 H(p/\beta q) < 0$ follows from inequality (42).

First, we shall find n_5. Assume that $k \le pn/2\beta$ in addition to the assumption that $k_0 \le k \le \alpha qn$. That is, assume that $k_0 \le k \le \min\{pn/2\beta, \alpha qn\}$. Since $H(z)$ is strictly monotone increasing over the interval $0 \le z \le 1/2$ and $(\lceil \beta k \rceil - 1)/pn < \beta k/pn \le 1/2$ by the assumption, we have

$$H\left(\frac{\lceil \beta k \rceil - 1}{pn}\right) \le H\left(\frac{\beta k}{pn}\right).$$

By inequality (60) and by the definitions of x, C_1, and F, we have

(63) $$nA(\beta, p, q, n, s, k) < \exp\left(nF(x) + \log n + C_1(\beta, p, q, s)\right).$$

Since if $0 < p/2\beta q < \alpha$ then $F(p/2\beta q) < 0$, for any $\varepsilon > 0$, we can take $n_3(\alpha, \beta, p, q, s, \varepsilon) > 0$ such that,

(64) for every integer $n \ge n_3$,
$$\exp\left(nF\left(\min\{\alpha, p/2\beta q\}\right) + \log n + C_1(\beta, p, q, s)\right) < \varepsilon.$$

Now we shall find an upper bound on $nF(k_0/qn)$. It follows from fact (57) and inequality (41) that

$$\begin{aligned} \text{(65)} \qquad \frac{F(z)}{z} &= q(1-sp)\frac{H(z)}{z} + \beta q \frac{H((\beta q/p)z)}{(\beta q/p)z} + C_2(\beta, p, q, s) \\ &< q(1+\beta-sp)\frac{H(z)}{z} + C_2(\beta, p, q, s) \end{aligned}$$

holds for any z with $0 < z \leq \alpha$. Moreover, it follows from the equation

$$\frac{H(z)}{z} = -\log z + \frac{1-z}{z}\log\frac{1}{1-z} = \log\frac{1}{z} + \log\left(1 + \frac{1}{(1-z)/z}\right)^{(1-z)/z}$$

that

$$\text{(66)} \qquad 0 < \log\frac{1}{z} < \frac{H(z)}{z} < 1 + \log\frac{1}{z} \quad \text{for any } z \text{ with } 0 < z < 1.$$

By inequality (44), we have

$$\text{(67)} \qquad 1+\beta-sp < 0.$$

It follows from fact (66), expression (65), and inequality (67) that

$$\text{(68)} \qquad \frac{F(z)}{z} < q(1+\beta-sp)\log\frac{1}{z} + C_2(\beta, p, q, s) \quad \text{for any } z \text{ with } 0 < z \leq \alpha.$$

If $k_0/qn \leq \alpha$, then we have

$$\begin{aligned} nF\left(\frac{k_0}{qn}\right) = \frac{k_0}{q}\frac{F(k_0/qn)}{k_0/qn} &< k_0(1+\beta-sp)\log\frac{qn}{k_0} + \frac{k_0}{q}C_2(\beta, p, q, s) \\ &= k_0(1+\beta-sp)\log n + C_3(\beta, p, q, s), \end{aligned}$$

where

$$C_3(\beta, p, q, s) = k_0(1+\beta-sp)\log\frac{q}{k_0} + \frac{k_0}{q}C_2(\beta, p, q, s).$$

Since $k_0(1+\beta-sp) \leq -2$ follows from the definition of k_0, for any $\varepsilon > 0$ we can take $n_4(\alpha, \beta, p, q, s, \varepsilon) > 0$ such that,

$$\begin{aligned} \text{(69)} \qquad &\text{for every integer } n \geq n_4, \\ &k_0/qn \leq \alpha \quad \text{and} \quad \exp\left(nF\left(k_0/qn\right) + \log n + C_1(\beta, p, q, s)\right) < \varepsilon. \end{aligned}$$

Thus, letting $n_5(\alpha, \beta, p, q, s) = \max\{n_3(\alpha, \beta, p, q, s, 1/2\alpha q), n_4(\alpha, \beta, p, q, s, 1/2\alpha q)\}$, statement (48) follows from fact (61) and inequalities (63), (64), and (69).

Next, we shall find n_7. Assume that $pn/2\beta < k$ in addition to the assumption that $k_0 \leq k \leq \alpha qn$. That is, assume that $pn/2\beta < k \leq \alpha qn$. It follows from inequality (60) and the definition of F that

$$\begin{aligned} &nA(\beta, p, q, n, s, k) \\ &\quad < \exp\left(nF(x) + \log n + C_1(\beta, p, q, s) + pn\left(H\left(\frac{\lceil \beta k \rceil - 1}{pn}\right) - H\left(\frac{\beta k}{pn}\right)\right)\right). \end{aligned}$$

It is easy to see that $H(z-d) \leq H(z) + H(d)$ for any $0 \leq z \leq 1$ and $0 \leq d \leq z$. Moreover, since $pn \geq 2$ follows from inequality (52), $H(z)$ is strictly monotone increasing over the interval $0 \leq z \leq 1/pn$. We therefore have

$$pn\left(H\left(\frac{\lceil \beta k \rceil - 1}{pn}\right) - H\left(\frac{\beta k}{pn}\right)\right) \leq pnH\left(\frac{\beta k - \lceil \beta k \rceil + 1}{pn}\right) \leq pnH\left(\frac{1}{pn}\right).$$

It follows from fact (66) that $pnH(1/pn) < 1 + \log p + \log n$. We therefore have

$$nA(\beta, p, q, n, s, k) < \exp(nF(x) + 2\log n + C_4(\beta, p, q, s)),$$

where $C_4(\beta, p, q, s) = 1 + \log p + C_1(\beta, p, q, s)$. Moreover, by fact (61) and the fact that $p/2\beta q < x = k/qn \leq \alpha$, which follows from the assumption that $pn/2\beta < k \leq \alpha qn$, we have

$$(70) \qquad nA(\beta, p, q, n, s, k) < \exp(n \max\{F(p/2\beta q), F(\alpha)\} + 2\log n + C_4(\beta, p, q, s)).$$

On the other hand, since it follows from fact (62) that $\max\{F(p/2\beta q), F(\alpha)\} < 0$, for any $\varepsilon > 0$ we can take $n_6(\alpha, \beta, p, q, s, \varepsilon) > 0$ such that,

(71) for every integer $n \geq n_6$,

$$\exp(n \max\{F(p/2\beta q), F(\alpha)\} + 2\log n + C_4(\beta, p, q, s)) < \varepsilon.$$

Thus, letting $n_7(\alpha, \beta, p, q, s) = n_6(\alpha, \beta, p, q, s, 1/2\alpha q)$, statement (49) follows from inequality (70) and fact (71).

Thus, we have found both n_5 and n_7, concluding the proposition.

Acknowledgments. We would like to thank the anonymous referee for the careful reading of the manuscript and helpful comments.

REFERENCES

[1] M. AJTAI, J. KOMLÓS, AND E. SZEMERÉDI, *An $O(n \log n)$ sorting network*, in Proc. 15th Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, (1983), pp. 1–9.

[2] ———, *Sorting in $c \log n$ parallel steps*, Combinatorica, 3 (1983), pp. 1–19.

[3] V. E. ALEKSEEV, *Sorting algorithms with minimum memory*, Kibernetica, 5 (1969), pp. 99–103.

[4] L. A. BASSALYGO, *Asymptotically optimal switching circuits*, Problemy Peredachi Informatsii, 17 (1981), pp. 206–211 (in Russian); English translation in Problems Inform. Transmission.

[5] D. E. KNUTH, *The Art of Computer Programming: Sorting and Searching*, vol. 3, Addison–Wesley, Reading, MA, 1973.

[6] M. S. PATERSON, *Improved sorting networks with $O(\log n)$ depth*, Algorithmica, 5 (1990), pp. 75–92.

[7] N. PIPPENGER, *Selection networks*, SIAM J. Comput., 20 (1991), pp. 878–887.

SIAM J. COMPUT.
Vol. 25, No. 4, pp. 740–774, August 1996

ANALYSIS OF BACKOFF PROTOCOLS FOR MULTIPLE ACCESS CHANNELS*

JOHAN HÅSTAD†, TOM LEIGHTON‡, AND BRIAN ROGOFF‡

Abstract. In this paper, we analyze the stochastic behavior of backoff protocols for multiple access channels such as the Ethernet. In particular, we prove that binary exponential backoff is unstable if the arrival rate of new messages at each station is $\frac{\lambda}{N}$ for any $\lambda > \frac{1}{2}$ and the number of stations N is sufficiently large. For small N, we prove that $\lambda \geq \lambda_0 + \frac{1}{4N-2}$ implies instability, where $\lambda_0 \approx .567$. More importantly, we also prove that any superlinear polynomial backoff protocol (e.g., quadratic backoff) is stable for any set of arrival rates that sum to less than one and any number of stations. The results significantly extend the previous work in the area and provide the first examples of acknowledgment-based protocols known to be stable for a nonnegligible overall arrival rate distributed over an arbitrarily large number of stations. The results also disprove a popular assumption that exponential backoff is the best choice among acknowledgment-based protocols for systems with large overall arrival rates. Finally, we prove that any linear or sublinear backoff protocol is unstable if the arrival rate at each station is $\frac{\lambda}{N}$ for any fixed λ and sufficiently large N.

Key words. Ethernet, backoff protocols, Markov chains, stochastic stability

AMS subject classifications. 60J10, 68M10, 90B12

1. Introduction.

Multiple access channels provide a simple and efficient means of communication in distributed systems. A typical example is the Ethernet [7], a local-area network where the channel consists of a tree made out of coaxial cable. When a station wants to send a message to one or more stations on the Ethernet, the sending station simply broadcasts the message throughout the entire system. Everyone, including the intended stations, then receives the message provided that there was no interference from other stations trying to send messages at the same time.

In order to reduce the chance of interference, stations check to make sure that the channel is clear before attempting to transmit a message. At first glance, it might seem that this precaution eliminates the possibility of a collision since the probability that two stations try to send at exactly the same instant in time is virtually zero. Unfortunately, collisions can still occur, since there is a nonnegligible delay between the time when a station begins to transmit and the other stations detect the transmission. Hence, if two or more stations attempt to transmit within this window of time, a collision will occur.

In the case of a collision, none of the messages is sent. Instead, the collision is detected and the messages are queued at their respective stations for retransmission at some point in the future. Of course, it would not make sense to retransmit right away since this would immediately result in another collision. Rather, packets are retransmitted according to a protocol that is often probabilistic in nature. For example, messages in an Ethernet are retransmitted again after T steps, where T is selected randomly from $\{1, 2, 3, \ldots, 2^{\min(10,b)}\}$ and b is the number of times the station has tried to send the packet but failed. This is one of a class of protocols generally referred to as *exponential backoff*.

The success of a protocol can be measured in several ways. For example, we might be interested in the average waiting time W_{ave} incurred by a message before it is successfully

*Received by the editors July 9, 1992; accepted for publication (in revised form) December 2, 1994. The research of this author was supported by Air Force Contract AFOSR-86-0078, Army Contract DAAL-03-86-K-0171, an NSF Presidential Young Investigator Award with matching funds from IBM, Xerox, and AT&T Bell Labs, and an IBM Postdoctoral Fellowship.

†Mathematics Department and Laboratory of Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139. Current address: Department of Numerical Analysis and Computing Science (NADA), Royal Institute of Technology (KTH), S-100 44 Stockholm, Sweden (johanh@nada.kth.se).

‡Mathematics Department and Laboratory of Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139 (ftl@math.mit.edu).

transmitted. Alternatively, we might consider the average number of waiting messages over time L_{ave} to be a better measure. Actually, these measurements are closely related. In fact, $L_{\text{ave}} = \lambda W_{\text{ave}}$ with probability one, where λ is the overall arrival rate of messages into the system over time [12].

For a protocol to be useful, it is crucial that $\mathrm{Ex}[L_{\text{ave}}]$ and $\mathrm{Ex}[W_{\text{ave}}]$ be small. In particular, we will want $\mathrm{Ex}[L_{\text{ave}}]$ and $\mathrm{Ex}[W_{\text{ave}}]$ to be finite. Note that this is a stronger condition than insisting only that L_{ave} and W_{ave} be finite with probability one. For example, consider the situation when $W_{\text{ave}} = 2^i$ with probability 2^{-i} for $i = 1, 2, 3 \ldots$.

Another measure of system performance that is often of interest to statisticians is the expected time $\mathrm{Ex}[T_{\text{ret}}]$ for the system to return to the *start state* (i.e., the state where all queues are empty). Of course, we will want this time to be as small as possible, and, in particular, we will want it to be finite.

Protocols for which $\mathrm{Ex}[L_{\text{ave}}]$, $\mathrm{Ex}[W_{\text{ave}}]$, and $\mathrm{Ex}[T_{\text{ret}}]$ are finite are said to be *stable*. Protocols for which all the measures diverge are said to be it unstable. Note that it is conceivable that there are protocols that are neither stable nor unstable as we have defined the terms here, since it might be that case that $\mathrm{Ex}[T_{\text{ret}}]$ is finite but $\mathrm{Ex}[W_{\text{ave}}]$ diverges for some protocol. However, all of the protocols considered in this paper are shown to be either entirely stable or entirely unstable in the sense defined above. In fact, the only reason we use these somewhat nonstandard definitions is that we want to encompass as many of the conflicting definitions of stability and instability in the literature as possible with our methods.

The *throughput rate* (i.e., the average rate of successful transmissions) is not a dominant concern. This is because the throughput rate is guaranteed to equal the arrival rate with probability one if the protocol is stable, but not vice-versa. As an example, consider a one-station system in which the sole station transmits with probability one if it has a message and in which the station receives a pair of new messages at each step with probability $\frac{1}{2}$. It is not difficult to show that with probability one, both the arrival and throughput rates for this system are one but that L_{ave} and W_{ave} diverge over time.

The development and analysis of transmission protocols that minimize average waiting time has been the subject of a great deal of work [1–7, 9–17]. We summarize some of this work in §2. Of greatest concern in this paper is the work on acknowledgment-based protocols. An *acknowledgment-based protocol* is one for which each station's transmission protocol is based only on its own history of successes and failures. In particular, the station is not assumed to have any knowledge of other stations' successes or failures or even of the number of stations in the system N.

Our present work is focused on a subset of acknowledgment-based protocols known as backoff protocols. A *backoff protocol* is one for which each station i containing a message transmits with probability $f(b_i)$, where f is a predetermined function and b_i (the *backoff counter* at station i) is the number of past consecutive failures by station i. After each successful transmission, b_i is reset to zero. After each failure, b_i is augmented by one. The value of b_i is left unchanged if no transmission is attempted.

Previous work has mostly centered on *binary exponential backoff* (for which $f(b) = 2^{-b}$), although other schemes such as *linear backoff* (for which $f(b) = \frac{1}{b+1}$) and *constant backoff* (for which $f(b)$ is simply a constant) have also been considered. Unfortunately, most of this work has been experimental and/or has depended on simplifying assumptions (e.g., that N is infinite) that render the consequences of any analysis less meaningful. Exceptions include some work on constant backoff (which serves as the basis for the Aloha protocol, but which is inherently unstable for fixed backoff and sufficiently large N) [13, 17], and the work of Goodman, Greenberg, Madras, and March [2], who proved that for any N, there is a λ' for which exponential backoff has finite $\mathrm{Ex}[T_{\text{ret}}]$ provided that the arrivals at station i are Bernoulli distributed with mean λ_i for $1 \leq i \leq N$, where $\lambda = \sum_{i=1}^{N} \lambda_i \leq \lambda'$. Unfortunately, λ' tends to zero as N increases and the question concerning stability for nonvanishing λ and large N remained open. In the case when $N = 2$, Goodman et al. also proved that exponential backoff has finite $\text{Ex}[T_{\text{ret}}]$ if λ_1 and λ_2 are at most 0.15 and infinite $\text{Ex}[T_{\text{ret}}]$ if $\lambda_1 > \frac{1}{2}$ and $\lambda_2 > 0$.

In this paper, we redirect the focus from exponential backoff to other protocols. Among other things, we show that exponential backoff is unstable whenever $\lambda_i \geq \frac{\lambda}{N}$ for $1 \leq i \leq N$ and $\lambda > 0.567 + \frac{1}{4N-2}$ or when $\lambda > \frac{1}{2}$ and N is sufficiently large. The result is not very surprising given the existing experimental data, but it does establish formal limits on the usefulness of exponential backoff. We also prove a much stronger and more important result concerning the stability of polynomial backoff protocols. In particular, we prove that if the arrivals at station i are Bernoulli distributed with rate λ_i then $f(b) = (b+1)^{-\alpha}$ backoff is stable for any constant $\alpha > 1$, any N and any $\{\lambda_i \mid \lambda = \sum_{i=1}^{N} \lambda_i < 1\}$. In terms of stability, the result is the strongest possible since any protocol is unstable if the overall arrival rate λ is one or larger. The result also provides the first example of an acknowledgment-based protocol known to be stable for nonvanishing λ and large N and proves that polynomial backoff protocols are superior to exponential backoff when λ is large.

The constraint that α be greater than one is crucial to the stability of polynomial backoff. In fact, we also prove that for any $\alpha \leq 1$, $f(b) = (b+1)^{-\alpha}$ backoff is unstable for any evenly distributed arrival rate λ and sufficiently large N.

Once a protocol has been found to be stable, the next step is to determine the precise values of $\text{Ex}[L_{\text{ave}}]$ and $\text{Ex}[W_{\text{ave}}]$. In particular, it is interesting to analyze the dependence of $\text{Ex}[L_{\text{ave}}]$ and $\text{Ex}[W_{\text{ave}}]$ on λ and N. Unfortunately, our current best estimates for these values are fairly weak. Whereas we do prove that $\text{Ex}[L_{\text{ave}}]$ and $\text{Ex}[W_{\text{ave}}]$ must grow polynomially as a function of N for almost any backoff protocol, we only upper bound this growth by an exponential function of N.

Quantifying the nature of an instability can also be of interest, particularly if the protocol is to be implemented in practice. In the cases of exponential backoff with high arrival rates and linear or sublinear backoff with large numbers of stations, we show that $\text{Ex}[L_{\text{ave}}]$ grows linearly over time, the worst possible scenario.

As a crucial aid in guiding our research, we performed computer simulations of several backoff protocols for various arrival rates and numbers of stations. Some of the data obtained can be found in §7. This data does suggest that quadratic backoff is a very competitive algorithm in practice. Linear backoff seems better if the number of stations is small or the load is minimal. Exponential backoff seems better only when the queues are massive.

The remainder of the paper is divided as follows. In §2, we describe the models for communication protocols more formally, introduce some further notation, and comment on the relevance of past research to our current work. In §3, we prove that polynomial backoff is stable for any arrival rate less than 1. In §4, we examine the dependence of L_{ave} and W_{ave} on λ and N. The instability of exponential backoff protocols is established in §5. In §6, we show that linear and sublinear backoff protocols are unstable for any fixed arrival rate and sufficiently large N. Section 7 contains some experimental data, and we conclude with some remarks and topics for research in §8.

2. Preliminaries.

2.1. Our model. In this paper, we follow the model of backoff protocols adopted in [2]. In this model, time is partitioned into equal-length intervals called *steps*. At the beginning of each step, a new message arrives at station i with probability λ_i for $1 \leq i \leq N$, where N is the number of stations in the system. The arrival of new messages is assumed to be independent over time and among stations. The *overall arrival rate* is defined to be $\lambda = \sum_{i=1}^{N} \lambda_i$. Arriving messages are added to the end of the *queues* located at each station. No limit is placed on the size of the queues, and if the system is unstable, they could become arbitrarily long over time.

The backoff protocol is governed by a function $f(x)$ defined in advance. At each step, the ith station attempts a transmission if it has a nonempty queue (allowing for the arrival of a new packet at the beginning of the step) with probability $f(b_i)$, where b_i is the value of the *backoff counter* at station i. For any set of b_i's, these probabilities are assumed to be independent over time and among stations. The backoff counters are initially zero. The ith counter is augmented by one whenever the ith station attempts to transmit but fails due to a collision. The ith counter is reset to zero whenever the ith station transmits successfully. If the ith station does not attempt to transmit, then the backoff counter is not changed. Confirmation of a collision or a successful transmission takes place during the same step in which the transmission was attempted. In addition, the message lengths are assumed to be smaller than the duration of a step.

The two important measures of efficiency are the average number of messages queued in the system at the end of each step L_{ave} and the average number of steps that each message must wait before it is sent W_{ave}. Since $L_{\text{ave}} = \lambda W_{\text{ave}}$ with probability 1, we will henceforth express our results in terms of L_{ave}.

2.2. Relevance of the model to reality. Our mathematical model differs from reality (e.g., the Ethernet) in several respects. We summarize these differences and their significance in the following paragraphs.

2.2.1. Upper bound on backoff counter. In the Ethernet, the backoff counter is never allowed to exceed a specified value $b_{\max}$, whereas in the mathematical model, it is allowed to become arbitrarily large. One mathematical problem with placing an upper bound on the backoff counter is that any such protocol becomes unstable for any fixed λ and large enough N. The reason is that for very large N, the system will eventually reach a state where almost every station has many messages queued. Once this happens, then with nonzero probability, the channel will become dominated by collisions and the throughput will be forever reduced to a trickle. The situation is less clear if the bound $b_{\max}$ is allowed to depend on N, but then individual stations would need to be informed about the number of other stations in the system. Of course, it might well be that reasonable upper bounds on N could be assumed in the computation of a bound $b_{\max}$ in practice. Research into such protocols for bounded N might prove to be interesting mathematically as well as useful in practice. For example, see [13, 17] and the references they contain for a discussion of constant backoff protocols.

2.2.2. Termination of undelivered messages. In the Ethernet, messages are discarded if they are not delivered within a specified amount of time. In our model, messages are never discarded and might be held in queue for an arbitrary amount of time. Discarding messages assures stability in the sense that $\text{Ex}[L_{\text{ave}}]$ is guaranteed to be finite, but only at the expense of discarding a nonzero fraction of the messages in systems that become unstable if discarding is not allowed.

2.2.3. Distribution of arrivals. In real systems, the distribution of arrivals may not be Bernoulli and may not be independent among stations. However, the Bernoulli and independence assumptions seem as reasonable as any others that are capable of being analyzed. Moreover, our analysis extends to several other natural distributions and can even be extended to systems where there is dependence among stations. For example, both stability and instability results hold for systems where at most two packets enter the entire system during a single step.

2.2.4. Selection of waiting time before attempted retransmission. In the Ethernet, the ith station attempts to rebroadcast t steps after the last attempt, where t is selected uniformly from $\{1, 2, 3, \dots, 2^{b_i}\}$. In our model, we retransmit at each step with probability 2^{-b_i}. The two methods for computing retransmission times are quite similar, but the former is easier to implement in practice (since it requires fewer random bits) and the latter is easier to analyze (since it is memoryless). It would seem unlikely that the stability results would differ for the two methods, but we have not proved this.

2.2.5. Message length. In reality, message lengths are much longer than the window during which conflicts can arise and be detected. Moreover, message lengths may vary from message to message. In the Ethernet, all messages are restricted to have the same length, and this length is a reasonably large multiple of the window of time used for conflict resolution. This difference between the Ethernet and our model is not as great as it might seem at first, however. The reason is that we can model a system where transmissions are long (but with uniform length) with a system where transmissions have zero length by simply compressing time to squeeze out transmission times altogether. This does not affect the conflict resolution process (which lies at the heart of stability analysis). Rather, we need only adjust the message arrival rates so that there are proportionately fewer arrivals in steps following nontransmissions. Although we will not go through the details here, it is not difficult to extend our stability results to hold for such a modified arrival process. Our instability results are also meaningful in systems with large message lengths, but require some changes. The reason is that we only know how to prove instability in a message-length-M model for arrival rate λ if the corresponding message-length-one model is unstable for arrival rate $\frac{\lambda}{\lambda+(1-\lambda)M}$. Hence, since our instability results for exponential backoff hold only for arrival rates exceeding 0.5, they only imply instability for arrival rates approaching one as M increases. Our linear and sublinear backoff results apply to any constant arrival rates and hence give the same results for any fixed M.

2.2.6. Synchronization. In real systems, time is not partitioned into discrete "windows" because there is no synchronization. In the Ethernet, a station that wants to transmit simply does so when and if the channel is clear locally. Nevertheless, it can be argued that our synchronous model accurately represents an asynchronous system to within a factor of two in window size [6]. Hence this assumption should not have a significant impact on stability analysis.

2.2.7. The bottom line. Whereas our model differs from reality in several notable respects, the differences are not all as important as they first seem. Moreover, the model is the most realistic among those that have been formally studied in the literature. In summary, the real contribution of this work is the development of formal techniques for analyzing communication protocols in multiple access channels, and the observation that protocols such as quadratic backoff may be superior to currently used protocols such as exponential backoff when the number of stations and/or the overall arrival rate is large.

2.3. Other models and results. Most of the work on protocols for communication in multiple access channels has focused on models for which the number of stations N is infinite [1, 3, 5, 9, 10, 14, 16]. The attraction for study of infinite models is clear. On the one hand, the analysis is simpler since with probability one, no two packets will ever arrive at the same station, and thus the disposition of packets is effectively independent of the station that transmits them (e.g., each message has its own backoff counter and is never contained in a queue with something else). On the other hand, there is the argument that the behavior of a protocol in an infinite model is reflective of its behavior in a real system with a large number of stations.

It has been our experience, however, that infinite model results often have fairly limited relevance to finite systems even when the number of stations is very large. Indeed, the reason is precisely that queuing plays a major role in any finite system (even one with large N) but is nonexistent in the infinite model. As a striking example, we note that Kelly proved in [5] that any polynomial backoff scheme is unstable in the infinite model. More recently, Aldous [1] extended this result to show that exponential backoff is also unstable in the infinite model. Whereas the complete disposition of exponential backoff in a finite model still remains unclear, we show that polynomial backoff is stable for any finite number of stations. Hence, the behavior of backoff protocols in infinite models can be misleading.

The study of infinite models has some implications for our paper, however. For example, the techniques used to prove the instability of backoff protocols in the infinite model can be used to prove that $\mathrm{Ex}[L_{\mathrm{ave}}]$ grows at least as a polynomial function of N in the finite model. More generally, it appears that a protocol is unstable in an infinite model if and only if the value of $\mathrm{Ex}[L_{\mathrm{ave}}]$ grows as a function of N in the corresponding finite model. In fact, we follow this strategy in proving lower bounds for $\mathrm{Ex}[L_{\mathrm{ave}}]$ in §4.

There has also been a great deal of work on models that require the use of more information when computing transmission probabilities. Some models use knowledge of the number of stations or try to approximate the number of stations that wish to transmit by analyzing the past history of the channel activity. Still others try to resolve conflicts among transmitting stations by using a playoff-type system to eventually choose a winner. Such schemes tend to be very stable for input rates up to a fixed threshold (e.g., $\frac{1}{e}$) but unstable for larger input rates. For examples of such models and their analysis see [4, 9, 16] and the references they contain.

As a final note, we point out that many protocols can be made stable for any fixed N and $\lambda < 1$ by simply allowing a transmitting station to empty its queue before allowing anyone else to start. Whereas such an approach may be necessary as a last ditch effort, it is not considered desirable since it allows a single station to dominate the system for a very long time, and since there must be a nontransmission step following the emptying of every queue. The latter constraint is particularly damaging in practice since if the protocol is working well, most queues will be very short, and hence the resulting frequency of nontransmissions is forced to be large (which means the protocol isn't working well after all). In particular, for large λ, we must have $\mathrm{Ex}[L_{\mathrm{ave}}] \geq \Omega(N)$ for such schemes. Although we prove an even greater asymptotic lower bound on $\mathrm{Ex}[L_{\mathrm{ave}}]$ for polynomial backoff, backoff protocols perform much better experimentally and are much simpler to implement. In fact, there is some reason to believe that polynomial backoff schemes perform like queue-emptying protocols during the rare times when they get into trouble (indeed, this possible behavior is the basis for our proof of stability), and otherwise behave similarly, but without the need for forced nontransmission steps.

2.4. Markov chains and their analysis. The performance of almost any protocol can be expressed in terms of the behavior of an associated Markov chain. For backoff protocols with a finite number of stations, we associate every possible configuration of backoff counters and queues $(\mathbf{b}, \mathbf{q}) = \{(b_1, \ldots, b_N, q_1, \ldots, q_N) \mid b_i \geq 0,\ q_i \geq 0 \text{ for } 1 \leq i \leq N\}$ with a unique state of the Markov chain. The initial state (or origin or zero) is identified with $(0, \ldots, 0)$. The associated infinite Markov chain is time invariant (the transition probabilities do not change with time), irreducible (every state is reachable from every other state) and aperiodic (the probability of being in any state at time t is positive if t is sufficiently large). We will below discuss some properties of Markov chains. For a more detailed discussion, we refer to [8].

A Markov chain is said to be *positive recurrent* if the expected time to return to zero $\mathrm{Ex}[T_{\mathrm{ret}}]$ is finite. It is said to be *transient* if the probability of returning to zero is less than one. Transience is a stronger condition than not being positively recurrent since any transient chain is clearly not positive recurrent, but not vice-versa. For example, an unbiased random walk is neither positive recurrent nor transient. We will also be interested in a third property, namely, whether the expected queue size $\mathrm{Ex}[L_{\mathrm{ave}}]$ over time is finite.

In the literature, a system is often said to be stable if it is positive recurrent. In practical situations, stability more naturally corresponds to the situation when $\mathrm{Ex}[L_{\mathrm{ave}}]$ is finite. Hence, we adopt a hybrid definition of stability in this paper. In particular, we say that a protocol is *stable* if it satisfies both conditions, and that it is *unstable* if it satisfies neither. Although there are hypothetical examples of systems that satisfy either condition but not both, the protocols we study either satisfy both or neither. Hence we will be able to classify protocols as stable in the strongest possible sense or unstable in an equally strong sense. Of course, there is also the possibility of the initial state being transient within the domain of instability. In fact, we conjecture that our instability results can be extended to prove the initial states is transient, although our techniques do not appear sufficient to prove such an extension.

The predominant method for analyzing the behavior of an infinite Markov chain is by means of a potential (or Lyapanov) function. In our case, a potential function is a map from $(\mathbb{Z}^+)^{2N}$ to $\mathbb{Z}^+$ such that the origin is mapped to zero and the other states are mapped to positive integers. Often, the potential function is directly tied to the measure of concern (e.g., the number of messages held in queues). For example, we will use potential functions of the form

$$c_1 \sum_{i=1}^{N} q_i \pm \sum_{i=1}^{N} f(b_i)^{-c_2} + c_3$$

for some constants $c_1 > 0$, $c_2 \geq 1$, and c_3.

The key step in proving that a protocol is unstable is to find a potential function for which the expected change in potential is at least δ for any state, where δ is a fixed positive constant. This, of course, implies that the expected value of the potential function after t steps is at least δt. By itself, this is not enough to imply instability. For example, consider the simple chain where state i moves to state $\max(i-1, 0)$ with probability $\frac{3}{4}$ and to state $i+1$ with probability $\frac{1}{4}$ and for which the potential of state i is $e^{2i} - 1$. This chain is positive recurrent but the expected change in potential is always at least 1.

If the potential function is natural enough, however, then such an argument can be used to prove instability. For example, we will use a potential function of the form

$$POT(\mathbf{q}, \mathbf{b}) = C \sum_{i=1}^{N} q_i + \sum_{i=1}^{N} f(b_i)^{-1} - N$$

for exponential backoff. If this potential function grows linearly with time, one can prove that $\mathrm{Ex}[L_{\mathrm{ave}}]$ diverges and that the system is unstable. We will prove this formally in §5.

The key step in proving that a protocol is stable is to find a potential function for which the expected change in potential is at most $-\delta$ for all but a finite number of states, where δ is a positive constant. Once such a potential function is found for $c_2 \geq 1$, it can then be shown that the associated chain is positive recurrent. To prove that $\mathrm{Ex}[L_{\mathrm{ave}}]$ is finite one has to study the expected change in POT^2. We will establish this connection in §3.

Unfortunately, it is not clear how to find such a potential function for polynomial backoff protocols. In fact, we suspect that there is no such potential function which increases monotonically with the q_i's and the b_i's. Hence, we must follow a somewhat more complicated approach to prove that polynomial backoff is stable.

In particular, we find a potential function for which there is a constant-depth tree of descendent states (not necessarily all of the same depth) emerging from each state for which the expected change in potential computed over these states is at most $-\delta$. In other words, the potential might be expected to increase in the first few steps but must decrease overall after some larger (but constant) number of steps. Such an argument is still sufficient to prove stability since the performance of such a chain is equivalent (up to constant factors) to the performance of a chain where each tree of descendent states is replaced by direct transitions from the root to the leaves with the appropriate probabilities. The latter chain is then shown to be positive recurrent and stable by the usual approach.

3. Stability of polynomial backoff. In this section, we show that polynomial backoff is stable under the most general assumptions. In particular, we prove the following theorem.

THEOREM 3.1. *Let $f(x) = (x+1)^{-\alpha}$ for any $\alpha > 1$. Then, for any number of stations N and any set of arrival probabilities $\lambda_1, \ldots, \lambda_N$ that sum to $\lambda < 1$, the backoff protocol defined by $f(x)$ is stable.*

The overall strategy of the proof is along the lines described in §2.4. In particular, we define the potential function

$$POT(\mathbf{q}, \mathbf{b}) = \sum_{i=1}^{N} q_i + \sum_{i=1}^{N} (b_i + 1)^{\alpha + \frac{1}{2}} - N,$$

where q_i is the length of the ith queue, b_i is the value of the ith backoff counter, and $\mathbf{q}$ and $\mathbf{b}$ are the corresponding vectors. We show that for every state with sufficiently large potential POT, there is a constant-depth tree of descendent states over which the expected decrease in POT^2 is at least δPOT for some fixed constant $\delta > 0$. By a tree of descendant states, we mean the following. Starting at a state $\mathbf{q}^0$, $\mathbf{b}^0$, we follow the system step by step. We observe the system and at each time we decide whether to halt the system or to let it run for another timestep. We always halt the system within a finite number of steps. The total set of halted states naturally forms the leaves of a tree and the maximal number of steps we observe the system is the depth of the tree. Standard theorems establishing convergence (see [8]) do not seem to apply to this situation, and hence to prove Theorem 3.1, we need the following lemma.

LEMMA 3.2. *Suppose that there are constants δ, d, and V such that for any state $(\mathbf{q}, \mathbf{b})$ which have potential $POT(\mathbf{q}, \mathbf{b}) \geq V$, there is a tree with depth at most d of descendent states over which the expected decrease in POT^2 is at least $\delta POT(\mathbf{q}, \mathbf{b})$. Let $T_{\mathrm{ret}V}$ denote the time at which the system returns to potential V or less (If $POT(\mathbf{q}, \mathbf{b}) \leq V$, then $T_{\mathrm{ret}V} = 0$). Then there is another constant c depending only on δ, d, and V for which*

$$Ex\left[\sum_{t=0}^{T_{\mathrm{ret}V}} L(t) | \, (\mathbf{q}^0, \mathbf{b}^0) = (\mathbf{q}, \mathbf{b})\right] \leq cPOT^2(\mathbf{q}, \mathbf{b}),$$

where $L(t)$ denotes the number of items in the system (i.e., total queue length*) at time t.*

Proof. We will prove the lemma by induction on time, but we have to be careful since $T_{\mathrm{ret}V}$ might be infinite a priori. To overcome the subtleties inherent in dealing with large values of $T_{\mathrm{ret}V}$, we define a modified system that is terminated after T steps. In particular, at time T, the system automatically returns to the origin and remains there forever. We then examine

$$E(\mathbf{q}, \mathbf{b}, T) = \mathrm{Ex}\left[\sum_{t=0}^{\min(T, T_{\mathrm{ret}V})} L(t) | \, (\mathbf{q}^0, \mathbf{b}^0) = (\mathbf{q}, \mathbf{b})\right]$$

and proceed by induction on time. For $T < 0$, we formally define $E(\mathbf{q}, \mathbf{b}, T) = 0$. Our induction hypothesis is

$$E(\mathbf{q}, \mathbf{b}, T) \leq cPOT^2(\mathbf{q}, \mathbf{b})$$

for all values of $\mathbf{q}$, $\mathbf{b}$, and T.

Provided that $c \geq 1$, the hypothesis is true for $T = 1$, since $L(0) \leq POT(\mathbf{q}, \mathbf{b})$ and $L(1) = 0$. In addition, the hypothesis is also true if $POT(\mathbf{q}, \mathbf{b}) \leq V$ by definition, since then $T_{\text{ret}V} = 0$. We next assume that

$$E(\mathbf{q}, \mathbf{b}, T') \leq cPOT^2(\mathbf{q}, \mathbf{b})$$

for all $T' < T$ and any $\mathbf{q}$ and $\mathbf{b}$ and consider the case when the system is terminated at time T.

Let the ith leaf of the tree of descendent states appear with probability p_i, have potential POT_i, and be at depth d_i. Also let L_i denote the sum of $L(t)$ over d_i steps taken to reach the ith leaf. Since at most one item be broadcast at any step and $L(t) \leq POT(\mathbf{q}^\mathbf{t}, \mathbf{b}^\mathbf{t})$, we can deduce $L(t-j) \leq POT(\mathbf{q}^\mathbf{t}, \mathbf{b}^\mathbf{t}) + j$ and hence

$$L_i \leq \sum_{j=0}^{d_i}(POT_i + j) = (d_i + 1)POT_i + \frac{d_i(d_i+1)}{2}.$$

Thus we can conclude that

$$\begin{aligned} E(POT, T) &\leq \sum_i p_i(L_i + E(POT_i, T - d_i)) \\ &\leq \sum_i p_i \left((d_i + 1)POT_i + \frac{d_i(d_i+1)}{2} + cPOT_i^2\right) \\ &\leq (d+1)\sum_i p_i POT_i + \frac{d(d+1)}{2} + c\sum_i p_i POT_i^2. \end{aligned}$$

By the assumption of the lemma, we know that

$$\sum_i p_i POT_i^2 \leq POT^2(\mathbf{q}, \mathbf{b}) - \delta POT(\mathbf{q}, \mathbf{b}).$$

A standard convexity argument can be used to show that this implies that

$$\sum_i p_i POT_i \leq POT.$$

Hence,

$$E(\mathbf{q}, \mathbf{b}, T) \leq (d+1)POT(\mathbf{q}, \mathbf{b}) + \frac{d(d+1)}{2} + cPOT^2(\mathbf{q}, \mathbf{b}) - c\delta POT(\mathbf{q}, \mathbf{b}).$$

By choosing $c \geq \frac{d+1}{\delta} + \frac{d(d+1)}{2\delta V}$, we can then conclude that $E(\mathbf{q}, \mathbf{b}, T) \leq cPOT^2(\mathbf{q}, \mathbf{b})$, which concludes the induction.

We have now proved that

$$\text{Ex}\left[\sum_{t=0}^{\min(T, T_{\text{ret}V})} L(t) | \, (\mathbf{q}^\mathbf{0}, \mathbf{b}^\mathbf{0}) = (\mathbf{q}, \mathbf{b})\right] \leq cPOT^2(\mathbf{q}, \mathbf{b})$$

for any T, where the constant c does not depend on T. Assume for the purposes of contradiction that

$$\text{Ex}\left[\sum_{t=0}^{T_{\text{ret}V}} L(t) | \, (\mathbf{q}^\mathbf{0}, \mathbf{b}^\mathbf{0}) = (\mathbf{q}, \mathbf{b})\right] > cPOT^2(\mathbf{q}, \mathbf{b})$$

for some state $(\mathbf{q}, \mathbf{b})$. Then there would be a finite T for which

$$\text{Ex}\left[\sum_{t=0}^{\min(T, T_{\text{ret}V})} L(t) | \, (\mathbf{q}^\mathbf{0}, \mathbf{b}^\mathbf{0}) = (\mathbf{q}, \mathbf{b})\right] > cPOT^2(\mathbf{q}, \mathbf{b}),$$

which is a contradiction. Hence

$$\mathrm{Ex}\left[\sum_{t=0}^{T_{\mathrm{ret}V}} L(t) | \, (\mathbf{q}^{\mathbf{0}}, \mathbf{b}^{\mathbf{0}}) = (\mathbf{q}, \mathbf{b})\right] \leq cPOT^2(\mathbf{q}, \mathbf{b}),$$

as claimed. □

Next we have the following result.

LEMMA 3.3. *Any system that satisfies the hypothesis of Lemma* 3.2 *and for which states with potential less than V can only move to states of potential at most $O(V)$ is stable.*

Proof. We use Theorem 14.0.1 of [8]. Let us state this theorem in our vocabulary.

THEOREM (Theorem 14.0.1 from [8]). *Given a Markov chain on a denumerable set which is irreducible and aperiodic and letting $f \geq 1$ be a function on its statespace, the following conditions are equivalent:*

(i) *The chain is positive recurrent with invariant probability measure π and the expected value for f with respect to π is finite.*

(ii) *There exist a finite set C of states such that*

$$\sup_{(\mathbf{q},\mathbf{b})\in C} \mathrm{Ex}\left[\sum_{t=1}^{T_{\mathrm{ret}C}} f(\mathbf{q}^{\mathbf{t}}, \mathbf{b}^{\mathbf{t}}) | \, (\mathbf{q}^{\mathbf{0}}, \mathbf{b}^{\mathbf{0}}) = (\mathbf{q}, \mathbf{b})\right] < \infty,$$

where $T_{\mathrm{ret}C} > 0$ is the time needed for the chain to return to C after step 0.

Lemma 3.3 follows from this theorem. We denote f as the total queue length and C as the set of states with potential at most V. Then condition (ii) follows from Lemma 3.2, and the conclusion of Lemma 3.3 is then given by (i). □

For polynomial backoff protocols with constant N and α, the potential of the system can increase by at most a constant factor at each step. Hence the condition of Lemma 3.3 that states with potential less than V can only move to states with potential $O(V)$ is easily seen to hold. Polynomial backoff protocols also satisfy the conditions of Lemma 3.2, although this is much harder to verify. In fact, the bulk of this section will be devoted to establishing the hypothesis of Lemma 3.2. The analysis is divided into four cases depending on the magnitude of the transmitting probabilities associated with the state. To this end, let us define

$$p_i = \begin{cases} \lambda_i & \text{if } b_i = 0 \text{ and } q_i = 0, \\ 1 & \text{if } b_i = 0 \text{ and } q_i > 0, \\ (b_i + 1)^{-\alpha} & \text{if } b_i > 0 \end{cases}$$

to be the probability that the ith station attempts a transmission. The four cases are then

(I) $\forall i \; b_i \leq B$,

(II) $\exists i \; b_i \geq B$ and $\forall i \; p_i < 1$,

(III) $\exists i \; b_i \geq B, \exists i \; p_i = 1$, and there exists another i with $p_i \geq \frac{1}{M}$, and

(IV) $\exists i \; b_i \geq B, \exists i \; p_i = 1$, and for all other i, $p_i \leq \frac{1}{M}$.

The values of B and M are constants to be defined later. Throughout, we will assume that $POT(\mathbf{q}, \mathbf{b}) \geq V$, where V is another large constant to be determined later.

In what follows, it will be convenient to let Q_i^+ (Q_i^-) denote the expected increase (decrease) in the potential due to changes in the length of the ith queue. We define B_i^+ and B_i^- analogously, and we let $Q^+ = \sum_{i=1}^N Q_i^+$, and define Q^-, B^+ and B^- analogously. We use $\mathrm{Ex}[X]$ denote the expected value of random variable X and $\Delta(P)$ to denote the amount change in the quantity P. For example, the expected change in potential will be written as $\mathrm{Ex}[\Delta(POT)]$. Lastly, we refer to the ith station as S_i. All these quantities are dependent on the present state $(\mathbf{q}, \mathbf{b})$ but due to readability considerations, we will not make this dependence explicit.

We analyze the cases in order of their difficulty. Case I is by far the most difficult and is saved for last. We start with Case II.

Case II. Without loss of generality, we can assume that $\forall i \; p_i < 1, b_1 \geq B$ and $b_i \leq b_1$ for $i > 1$.

We consider a single step of the system and analyze $\Delta(POT^2)$. By definition,

$$\begin{aligned}\mathrm{Ex}[\Delta(POT^2)] = {} & \mathrm{Ex}[\Delta(POT^2) | S_1 \text{ succeeds}]\Pr[S_1 \text{ succeeds}] \\ & + \mathrm{Ex}[\Delta(POT^2) | S_1 \text{ does not succeed}]\Pr[S_1 \text{ does not succeed}].\end{aligned}$$

When S_1 succeeds, the potential decreases by at least

$$(b_1 + 1)^{\alpha+\frac{1}{2}} - N \geq \frac{1}{2}(b_1 + 1)^{\alpha+\frac{1}{2}}$$

if $B \geq 10N$. This in turn corresponds to a decrease in magnitude for POT^2 of at least

$$(b_1 + 1)^{\alpha+\frac{1}{2}} POT - \frac{1}{4}(b_1 + 1)^{2\alpha+1} \geq \frac{1}{2}(b_1 + 1)^{\alpha+\frac{1}{2}} POT$$

since $POT \geq \frac{1}{2}(b_1 + 1)^{\alpha+1/2}$ by definition.

The probability of S_1 succeeding is at least

$$(b_1 + 1)^{-\alpha} \prod_{i=2}^{N} (1 - p_i) \geq (1 - \lambda)2^{-N}(b_1 + 1)^{-\alpha}.$$

Thus the first term in the expression for $\mathrm{Ex}[\Delta(POT^2)]$ is at most

$$-\frac{1}{2}(1 - \lambda)2^{-N}(b_1 + 1)^{\frac{1}{2}} POT.$$

To estimate the second term, we use $\Delta Q^+ \leq N$, $\Delta Q^- \geq 0$, and $\Delta B^- \geq 0$ to obtain

$$\begin{aligned}& \mathrm{Ex}[\Delta(POT^2) | S_1 \text{ does not succeed}]\Pr[S_1 \text{ does not succeed}] \\ & \leq \mathrm{Ex}[\Delta(POT^2) | S_1 \text{ does not succeed}] \\ & \leq \mathrm{Ex}[(POT + N + \Delta B^+)^2] - POT^2 \\ & \leq N^2 + 2N \cdot POT + 2(POT + N)\mathrm{Ex}[\Delta B^+] + \mathrm{Ex}[(\Delta B^+)^2],\end{aligned}$$

where ΔB^+ is conditional upon the fact that S_1 does not succeed. Note that this does not mean that S_1 tried and failed, since S_1 probably did not even try. In any event, if is not difficult to verify that $\mathrm{Ex}(\Delta B^+) \leq N$ and $\mathrm{Ex}((\Delta B^+)^2) \leq N(b_1 + 1)^{\alpha-1} + N^2$. Plugging in and summing, we find that the second term is at most

$$4N^2 + 4N \cdot POT + N(b_1 + 1)^{\alpha-1}.$$

Combining the two terms with the inequality $POT \geq (b_1 + 1)^{\alpha-1}$ then gives

$$\mathrm{Ex}[\Delta(POT^2)] \leq 4N^2 + 4N \cdot POT + N(b_1 + 1)^{\alpha-1} - \frac{1}{2}(1 - \lambda)2^{-N}(b_1 + 1)^{\frac{1}{2}} POT \leq -\delta POT$$

provided that $\delta < 1$, $POT \geq N$, and

$$B \geq \frac{c_B N^2 2^{2N}}{(1-\lambda)^2}$$

for some constant c_B independent of N and λ.

Case III. We assume $b_1 \geq B,\ p_2 = 1,\ p_j \geq \frac{1}{M},\ j \neq 2$, and $b_i \leq b_1$ for $i > 1$.

We will proceed as in Case II, except that here we analyze the expected change in POT^2 over two steps instead of one. The most desirable scenario is when station j crashes with station 2 in the first step while no station with $b_i > N$ attempts, and station 1 is the only station to transmit at the second step. Call this event E. Then

$$\mathrm{Ex}[\Delta(POT^2)] = \mathrm{Ex}[\Delta(POT^2)|E]\mathrm{Pr}[E] + \mathrm{Ex}[\Delta(POT^2)|\neg E]\mathrm{Pr}[\neg E].$$

The decrease in potential when E happens is at least

$$(b_1+1)^{\alpha+\frac{1}{2}} - M - N^{\alpha+\frac{3}{2}} \geq \frac{1}{2}(b_1+1)^{\alpha+\frac{1}{2}}$$

provided $B \geq \max(M, N^2)$. The probability of E is at least

$$\frac{1}{2}p_j \frac{1}{(b_1+1)^\alpha} \prod_{i=2}^{N}(1-p_i'),$$

where p_i' is the value of p_i after the first step as prescribed above. Reasoning as in Case II, we can then conclude that the first term is at most $-\frac{1}{4}(1-\lambda)(b_1+1)^{1/2}\frac{1}{M}2^{-N}POT$.

We have the same estimates for the second term as in Case II and this gives the desired conclusion provided that

$$B \geq \frac{c_B N^2 M^2 2^{2N}}{(1-\lambda)^2}$$

for some constant c_B independent of N and λ.

Case IV. We assume $p_1 = 1; \forall i > 1\ p_i \leq \frac{1}{M}$.

In the last two cases, we can simplify the analysis for $\mathrm{Ex}[\Delta(POT^2)]$ by finding bounds on $\mathrm{Ex}[\Delta POT]$ and $\mathrm{Ex}[(\Delta POT)^2]$ for some tree of descendent states. In particular, a simple calculation reveals that

$$\mathrm{Ex}[\Delta(POT^2)] = 2POT \cdot \mathrm{Ex}[\Delta POT] + \mathrm{Ex}[(\Delta POT)^2]$$

for any set of descendent states, and hence we can prove that $\mathrm{Ex}[\Delta(POT^2)] \leq -\delta POT$ by showing that $\mathrm{Ex}[\Delta POT] \leq -\delta$ and $\mathrm{Ex}[(\Delta POT)^2] \leq \delta POT$. We start by bounding $\mathrm{Ex}[\Delta POT]$. In this case, we need only consider one step of the system. Proceeding as in Cases II and III, we find that $Q_i^+ = \lambda_i$,

$$Q_1^- \geq \left(1 - \frac{1}{M}\right)^{N-1} \geq 1 - \frac{N}{M},$$

$$B_i^+ \leq \begin{cases} O(\frac{N}{M}) & \text{if } i = 1, \\ O(M^{-1/(2\alpha)}) & \text{if } i > 1, \end{cases}$$

and $B_i^- = 0$. Hence

$$\mathrm{Ex}[\Delta POT] \leq \lambda - 1 + \frac{N}{M} + O(NM^{-1/(2\alpha)}) \leq -\delta$$

provided that $\lambda < 1, \delta < \frac{1-\lambda}{2}$, and

$$M \geq \frac{c_M N^{2\alpha}}{(1-\lambda)^{2\alpha}}$$

for some constant c_M independent of N and λ.

To finish the argument, we estimate $\mathrm{Ex}[(\Delta POT)^2]$ as follows:

$$\begin{aligned}\mathrm{Ex}[(\Delta POT)^2] &\leq 4\mathrm{Ex}[(\Delta Q^+)^2] + 4\mathrm{Ex}[(\Delta Q^-)^2] + 4\mathrm{Ex}[(\Delta B^+)^2] + 4Ex[(\Delta B^-)^2] \\ &\leq 4N^2 + 4 + O\left(\frac{N}{M} + NM^{1-1/\alpha}\right) \\ &\leq \delta POT\end{aligned}$$

for $POT \geq V$ where $V \geq \frac{c_V}{\delta} N^2 M$ and c_V is a constant that is independent of N and λ.

Case I. $\forall i\ b_i \leq B$.

We will proceed as in Case IV. In particular, the bulk of the proof is devoted to showing that $\mathrm{Ex}[\Delta POT] \leq -\delta$. Afterwards, we observe that $\mathrm{Ex}[(\Delta POT)^2] \leq \delta POT$.

By making V large enough (the exact value will be determined later) and noting that if $POT \geq V$, we can assume that $q_N \geq \frac{V}{N} - (B+1)^{\alpha+1/2}$ and $b_N \leq B$ without loss of generality. In other words, the Nth queue is very large and accounts for a good proportion of the overall potential.

The key to the proof is to show that with some not-too-small probability, the Nth station effectively dominates the channel for a very long time, thereby substantially reducing its massive queue and dramatically lowering the overall potential function. In particular, we show that there is a not-too-small probability that S_N is always the sole next station to broadcast after any collision. This is the hard part of the argument. Once this is done, we finish up by showing that there aren't too many collisions over time and that not too many packets arrive over time. Of course, we must be sure to check that things can't get too bad if the Nth station ever does lose control.

To prove that we have a small probability of the Nth station staying in control, we will first study what happens to a system of $N-1$ stations when we assume that all transmissions fail. This is essentially the situation when the Nth station never loses control.

LEMMA 3.4. *Consider an isolated system where a single station advances from level i to $i+1$ with probability $i^{-\alpha}$ and otherwise remains at level i. Suppose the initial level of the station is between S and B. Then with probability exceeding $1 - O(2^{-c\sqrt{S}})$, the station reaches level b (for any $b \geq S$) within time $6\alpha b^{\alpha+1}$, and the station moves from level b to $b+1$ after time $b^{\alpha+1}/4e^{\alpha+1}$ for any $b \geq 2B$.*

Proof. Without loss of generality, we assume that the station starts at level S for the first part of the proof and that it starts at level B for the second part. To avoid duplication of effort in the proof, we will use R to denote either S or B.

We start by computing the probability $\Pr[b, t]$ that the transition from b to $b+1$ is made at step t. This probability is precisely

$$\Pr[b,t] = \sum_{\substack{t_R+\cdots+t_b=t-r,\\ t_j\geq 0}} \left(\prod_{j=R}^{b}(1-j^{-\alpha})^{t_j} j^{-\alpha}\right),$$

where $r = b+1-R$ and t_j denotes the number of steps that started and ended with the station in level j for $R \leq j \leq b$. Using the inequality $1 - x \leq e^{-x}$ and simplifying, we find that

$$\Pr[b,t] \le \prod_{j=R}^{b} \frac{1}{j^{\alpha}} \sum_{\substack{t_R+\cdots+t_b=t-r,\\ t_j \ge 0}} e^{-\sum_{j=R}^{b} t_j/j^{\alpha}}.$$

Since $\sum_{j=R}^{b} t_j = t - r$ and $t_j \ge 0$ for $R \le j \le b$, it is clear that $\sum_{j=R}^{b} \frac{t_j}{j^{\alpha}} \ge \frac{t-r}{b^{\alpha}}$. Hence

$$\begin{aligned}\Pr[b,t] &\le \frac{(R-1)!^{\alpha}}{b!^{\alpha}} e^{-(\frac{t-r}{b^{\alpha}})} \left(\sum_{\substack{t_R+\cdots+t_b=t-r,\\ t_j \ge 0}} 1 \right) \\ &\le \frac{(R-1)!^{\alpha} e^{r/b^{\alpha}}}{b!^{\alpha} e^{t/b^{\alpha}}} \binom{t}{r} \le \frac{R^{\alpha R-\alpha} e^{\alpha b} e^{r/b^{\alpha}} t^r e^r}{b^{\alpha b} e^{\alpha R-\alpha} e^{t/b^{\alpha}} r^r} \le \frac{e^{(\alpha+1)r} e^{r/b^{\alpha}} t^r}{b^{\alpha r} e^{t/b^{\alpha}} r^r} \\ &= \left(\frac{e^{\alpha+1} t e^{1/b^{\alpha}}}{b^{\alpha} r e^{t/rb^{\alpha}}} \right)^r.\end{aligned}$$

In order to bound the behavior of this function, it is most useful to let $\beta = \frac{t}{b^{\alpha} r}$. Then

$$\Pr[b,t] \le \left[\frac{e^{\alpha+1} e^{1/b^{\alpha}} \beta}{e^{\beta}} \right]^r.$$

For large or small constant values of β, the preceding expression is very small. In particular, for $\beta \le 1/2e^{\alpha+1}$,

$$\Pr[b,t] \le \left(\frac{e^{1/b^{\alpha}}}{2e^{1/2e^{\alpha+1}}} \right)^r \le \frac{1}{2^r},$$

assuming $b^{\alpha} \ge 2e^{\alpha+1}$, which will always be true since $b \ge R$. There are at most $b^{\alpha} r/2e^{\alpha+1}$ values of $t \le b^{\alpha} r/2e^{\alpha+1}$. Hence, the probability that we progress from b to $b+1$ before step $b^{\alpha} r/2e^{\alpha+1} \le b^{\alpha+1}/2e^{\alpha+1}$ is at most

$$\frac{b^{\alpha} r}{2e^{\alpha+1} 2^r} \le \frac{b^{\alpha+1} 2^R}{2e^{\alpha+1} 2^b} \le O\left(\frac{b^{\alpha+1} 2^R}{2^b} \right),$$

and thus with $R = B$, $b \ge 2B$, we have established the second part of the lemma.

For $\beta \ge 6\alpha$, the bound is at most

$$\left[\frac{6\alpha e^{1/b^{\alpha}} e^{\alpha+1}}{e^{6\alpha}} \right]^r \le \frac{1}{2^r}.$$

This is small for $r \ge \sqrt{S}$. For smaller r, we need to observe that $\beta = t/b^{\alpha} r \ge \sqrt{S}$ for $t = 6\alpha b^{\alpha+1}$, $b \ge S$, and $r \le \sqrt{S}$, and in this case we use the bound

$$\Pr[b,t] \le \left[\frac{e^{\alpha+1} e^{1/b^{\alpha}} \beta}{e^{\beta}} \right]^r \le e^{-c\sqrt{S}}$$

for some constant c. Moreover, the bound forms a geometric series for $\beta \ge 6\alpha$. Thus the probability that the transition from b to $b+1$ is made after step $6\alpha b^{\alpha} r$ is at most $O(2^{-c\sqrt{S}})$ for $r \le \sqrt{S}$, and for $r \ge \sqrt{S}$, we have the bound

$$O\left(\frac{b^{\alpha} r \alpha}{2^r} \right) \le O\left(\frac{b^{\alpha+1} 2^R}{2^b} \right).$$

Summing over b again gives a geometric series, and for $r \geq \sqrt{S}$, we get the total estimate

$$1 - O\left(2^{-c\sqrt{S}}\right),$$

and the first part of the lemma is also established. □

Lemma 3.4 can be immediately extended to hold for $N-1$ isolated stations simultaneously by simply adding the failure probabilities. In other words, the result holds for $N-1$ stations simultaneously with probability exceeding $1 - O(N2^{-c\sqrt{S}})$.

Having established how the rest of the system behaves if the Nth station remains in control, we next look at the chances that the Nth station does maintain control.

LEMMA 3.5. *Suppose S_N collides with another station at time T and backs off to $b_N = 1$. Then the probability that S_N will transmit successfully before any other station attempts a transmission is at least* $1 - 2^{\alpha} \sum_{i=1}^{N-1} p_i$.

Proof. Let $W = \prod_{i=1}^{N-1}(1-p_i)$ be the probability that none of the first $N-1$ stations try to send on a given step. Then the probability that the Nth station continues to maintain control after the collision is at least

$$2^{-\alpha}W + (1-2^{-\alpha})2^{-\alpha}W^2 + (1-2^{-\alpha})^2 2^{-\alpha}W^3 + \cdots$$
$$= \frac{2^{-\alpha}W}{1-(1-2^{-\alpha})W} = \frac{W}{2^{\alpha}-(2^{\alpha}-1)W}.$$

Replacing W with $1-\epsilon$, we observe that the probability of maintaining control is at least

$$\frac{1-\epsilon}{1+\epsilon(2^{\alpha}-1)} \geq 1 - 2^{\alpha}\epsilon.$$

Hence the probability of not regaining control at the next transmission is at most

$$2^{\alpha}\epsilon = 2^{\alpha}(1-W) \leq 2^{\alpha}\sum_{i=1}^{N-1} p_i,$$

since

$$W = \prod_{i=1}^{N-1}(1-p_i) \geq 1 - \sum_{i=1}^{N-1} p_i,$$

and the lemma follows. □

We now use Lemmas 3.4 and 3.5 to prove that S_N has a not-too-small probability of remaining in control for a very long time. The basic idea is that S_N keeps successfully transmitting until a collision occurs, whereupon it regains control before anyone else attempts to transmit. We will consider two kinds of collisions. The first involves collisions with stations that have backoff counters of size S or larger, and the second involves a collision with stations that have backoff counters of size less than S. There is also the possibility of both kinds of collisions happening simultaneously, but we will rig things so that this does not happen. By this we mean that the good set of events in which S_N remains in control this will not happen.

Collisions of the first kind are nice because the behavior of stations with backoff counters of size S or larger is governed by Lemma 3.4. Collisions of the second kind are nice because there are not very many of them, provided that we never allow the first $N-1$ stations to transmit successfully. In what follows, we consider sequences of events for which the Nth station always maintains control by directly blocking transmissions for other stations. We will show that no matter what times are chosen for the attempted transmissions of stations with small backoff counters, there is a not-too-small probability that everything works as we hope.

To start things off, we consider the probability that S_N succeeds in the first or second step. For this to happen, we need S_N to broadcast at the first step to block anybody else from succeeding. We also keep anyone else from broadcasting at the second step so that S_N can succeed and establish control. This sequence of events happens with probability at least

$$\Omega(B^{-\alpha}(B+1)^{-\alpha}(1-\lambda)2^{-N}).$$

Henceforth, we will consider only sequences that started in this fashion and thus have $b_N = 0$ at step 3.

Next define σ_γ to be the set of times (excluding steps 1 and 2) that one of the first $N-1$ stations would have made an attempt to transmit with backoff counter less than S if all its previous transmissions would have failed. Our argument will allow any possible configurations of σ_γ, observe only that by definition that $|\sigma_\gamma| \leq (N-1)S$. We partition σ_γ into $k \leq (N-1)S$ maximal intervals $I_1, I_2, \ldots, I_k$ of configuration steps, and we define T_i to be the step following I_i for $1 \leq i \leq k$. By definition, $T_i \notin \sigma_\gamma$ for $1 \leq i \leq k$. Lastly, set $\sigma'_\gamma = \sigma_\gamma \cup \{T_i | 1 \leq i \leq k\}$.

At each step of σ'_γ, we will require that S_N attempts a transmission and that each station with backoff counter S or larger does not attempt a transmission. This will ensure that stations with small backoff counters never succeed and that S_N regains control after a collision with any such station. Provided that the Nth station otherwise retains control (i.e., that $b_N \leq 1$ before each I_i), the probability that these forced moves actually take place is at least

$$(2NS)!^{-\alpha}(1-S^{-\alpha})^{2N^2S} \geq \frac{1}{2}(2NS)!^{-\alpha},$$

provided that $S \geq (4N^2)^{\frac{1}{\alpha-1}}$. The $(2NS)!^{-\alpha}$ factor is a gross underestimate on the probability that S_N transmits at all the desired times (which could all be bunched together in one large interval), and the $(1-S^{-\alpha})^{2N^2S}$ factor accounts for the probability that the stations with large backoff counters do not attempt to transmit at all the desired times.

The preceding analysis accounts for collisions with stations that have small backoff counters. To account for stations that have large backoff counters, we apply Lemmas 3.4 and 3.5. In particular, we let E_t denote the event "At time t, S_N collides with another station, backs off to $b_N = 1$ and does not regain control by being the next station to send". If S_N loses control E_t must happen for some t. We need only analyze what happens outside σ'_γ, so S_N will only compete with stations with large backoff counters. By Lemma 3.5, we have

$$\Pr[E_t] \leq 2^\alpha \left(\sum_{b_i \geq S}(b_i(t)+1)^{-\alpha}\right)^2 \leq 2^\alpha\,(N-1)\sum_{b_i \geq S}(b_i(t)+1)^{-2\alpha},$$

where the second inequality follows by Cauchy–Schwartz inequality.

We next need to sum $\Pr[E_t]$ over $t \notin \sigma'_\gamma$. To bound this probability, we will assume that the conclusion of Lemma 3.4 holds at time t but not necessarily at any future time so as to avoid conditioning of the probabilities. We also have to be careful to note that the value of $b_i(t)$ depends on when S_i first had a backoff counter of size S, but otherwise is governed by Lemma 3.4. Combining these observations gives

$$
\begin{aligned}
\sum_{t \notin \sigma'_\gamma} \Pr[E_t] &\leq 2^\alpha (N-1) \sum_{t \notin \sigma'_\gamma} \sum_{b_i \geq S} (b_i(t)+1)^{-2\alpha} \\
&\leq 2^\alpha (N-1) \sum_{i=1}^{N-1} \sum_{t \notin \sigma'_\gamma, b_i(t) \geq S} (b_i(t)+1)^{-2\alpha} \\
&\leq 2^\alpha (N-1)^2 \sum_{t=1}^{\infty} \left(\max \left(S, \left(\frac{t}{6\alpha} \right)^{\frac{1}{\alpha+1}} \right) \right)^{-2\alpha} \\
&\leq 2^\alpha (N-1)^2 \left(\frac{6\alpha S^{\alpha+1}}{S^{2\alpha}} + \sum_{t=6\alpha S^{\alpha+1}}^{\infty} \left(\frac{t}{6\alpha} \right)^{\frac{-2\alpha}{1+\alpha}} \right) \\
&\leq c N^2 S^{1-\alpha},
\end{aligned}
$$

assuming that the conclusion of Lemma 3.4 holds.

Although we still have many details to check, we are essentially done with the hard part of the analysis. In what follows, we consider descendent states with depth at most $U + 2NS$, where $U + 2NS \leq q_N$. In particular, we are interested in sequences of descendent states for which the following conditions hold:

(1) every backoff counter is at most $O(U^{1/(\alpha+1)})$;

(2) S_N successfully broadcasts for all but $O(NU^{1/(\alpha+1)} \log U + NS)$ steps;

(3) the number of new messages arriving overall in the first T steps is at most $\lambda T + 2NS + U^{1/2} \log U$ for all $T \leq U$; and

(4) S_N gains control in the first two steps and maintains control thereafter (i.e., the conditions described in the previous discussion are satisfied).

We first note that if all of these conditions hold for U steps, then we will have experienced a tremendous decrease in the potential function. This is because at least $U - O(NU^{1/(\alpha+1)} \log U)$ messages are successfully transmitted, at most $\lambda U + U^{1/2} \log U + 2N^2 S$ arrive, and each backoff counter adds at most $O(U^{(\alpha+1/2)/(\alpha+1)})$ to the potential. Hence the decrease in potential is at least

$$
\begin{aligned}
&(1-\lambda)U - O\left(NU^{\frac{\alpha+1/2}{\alpha+1}} + U^{\frac{1}{2}} \log U + N^2 S\right) \\
&\quad = (1-\lambda)U - O\left(NU^{\frac{\alpha+1/2}{\alpha+1}}\right),
\end{aligned}
$$

which is large for large U.

We next note that the probability that all of these conditions hold for U steps is not too small. This follows naturally form the proceeding analysis and Lemmas 3.4 and 3.5. In particular, the probability of gaining control in the beginning is

$$\Omega(B^{-2\alpha}(1-\lambda)2^{-N}).$$

Given that S_N gains control by the method described at the beginning and that $b_N \leq 1$ at steps before I_i $(1 \leq i \leq k)$, the probability of having things go as planned for steps in σ'_γ is $\Omega((2NS)!^{-\alpha})$. Given that things have gone well at the beginning and during the previous steps of σ'_γ, the probability of not violating Lemma 3.4 nor having S_N otherwise lose control is at least

$$1 - O(N2^{-c\sqrt{S}} + N^2 S^{1-\alpha}).$$

The first term comes from Lemma 3.4 and the second comes from Lemma 3.5 and the above calculation. Thus we have calculated the probability of (4) holding. The probability of violating the first condition is $O(2^{-cU^{1/2(\alpha+1)}})$ by Lemma 3.4. If the first condition is satisfied and S_N remains in control, we know that there are at most $O(NU^{1/(\alpha+1)} + NS)$ collisions in the U steps. The time for the S_N to try to send again after each collision is at most $a \log U$ with probability $1 - O((NU^{1/(\alpha+1)}(1-2^{-\alpha})^{a \log U})$. This is very small for $a = 2^{\alpha+1}$. Since the next attempted transmission will always be successful with probability $1 - O(N^2/S^{\alpha-1})$, we can conclude that the Nth station successfully transmits on all but $O((NU^{1/(\alpha+1)} + NS) \log U)$ steps with probability $1 - O(N^2/S^{\alpha-1} + \frac{N}{U})$. Hence with this probability, condition (2) is satisfied. The last condition is easily verified to hold with probability $1 - O(\frac{1}{U})$ by standard arguments.

Putting all the probabilities together, we find that all desired conditions hold with probability exceeding

$$\Omega(B^{-2\alpha}(1-\lambda)2^{-N}(2NS)!^{-\alpha})\left(1 - O\left(N2^{-c\sqrt{S}} + N^2S^{1-\alpha} + \frac{N}{U}\right)\right)$$
$$\geq \Omega(B^{-2\alpha}(1-\lambda)2^{-N}(2NS)!^{-\alpha})$$

for $U \geq c_U N$ and $S \geq c_S N^{\frac{2}{\alpha-1}}$.

Note that we have multiplied probabilities of success instead of adding probabilities of failure at two crucial points of the analysis. The first place we do this is at the beginning, when we force S_N to gain control right away. Since later probabilities are conditioned upon this happening, multiplication of success probabilities for the first two steps and later steps is appropriate. The second place we multiply success probabilities is when we combine the probabilities that things work well during σ'_γ with the probability that things work well outside σ'_γ. Although these probabilities are not completely independent, the dependence is minimal and works in our favor. This is formally argued as follows.

Let $\rho_{i,t}$ and $\rho^*_{i,t}$ be random numbers drawn uniformly and independently from [0, 1] for $1 \leq i \leq N$ and $3 \leq t \leq U + 2NS$. The value of $\rho^*_{i,t}$ will be compared with λ_i; to decide it, the ith station gets a new packet at time t, and $\rho_{i,t}$ will be compared with $b_i(t)^{-\alpha}$ to decide if the ith station tries to transmit at the tth step. Note that the values of $b_i(t)$ depend on previous values of $\rho_{i,t}$ and $\rho^*_{i,t}$ for various i's and t's but that the ρ-values are mutually independent.

For each S_i, examine the values of $\rho_{i,t}$ and $\rho^*_{i,t}$ for $3 \leq t \leq U + 2NS$ to determine the steps (if any) at which S_i would try to transmit with backoff counter less than S under the *assumption* (not the knowledge) that all attempts are blocked. Accumulating these values for $1 \leq i \leq N-1$, determine the time steps contained in σ'_γ. Note that the selection of steps that are in σ'_γ is independent of values of $\rho_{i,t}$ and $\rho^*_{i,t}$ for i such that $i = N$ or S_i has a backoff counter of size S or larger at step t under the assumption that all previous attempts have been blocked. Moreover, the remainder of this argument will not depend in any way on what steps were selected for σ'_γ.

We now analyze the probability that $\rho_{i,t}$ and $\rho^*_{i,t}$ are as we would hope for $t \in \sigma'_\gamma$ and i such that $i = N$ or S_i has a large backoff counter. In particular, we want S_N to try to broadcast for all $t \in \sigma'_\gamma$, and we do not want stations with a large backoff counter to try to transmit at any step $t \in \sigma'_\gamma$. As argued before, the ρ-values satisfy these demands with probability $\Omega((2NS)!^{-\alpha})$, provided only that $b_N \leq 1$ at the beginning of each I_i, $1 \leq i \leq k$. In addition, we can have at most N^2S new arrivals during σ'_γ, and S_N is thwarted from broadcasting for at most NS steps during these times.

We next consider the $\rho_{i,t}$ and $\rho^*_{i,t}$ values for $t \notin \sigma'_\gamma$ and i such that $i = N$ or S_i has a large backoff counter. To simplify the argument, we first consider the behavior of the system as if σ'_γ did not exist. In this scenario, we can apply Lemmas 3.4 and 3.5 and the analysis that followed to conclude that with probability $1 - O(N2^{-c\sqrt{S}} + N^2S^{1-\alpha} + \frac{N}{U})$, the values for $\rho_{i,t}$ and $\rho^*_{i,t}$ make the system perform exactly as desired. In other words, the values of the large backoff counters are regulated by Lemma 3.4, S_N never loses control, new packets do not arrive too fast, etc. Note that we add the probabilities of failure in this context since the various modes of failure in the isolated system might be dependent. Also note that stations with small backoff counters are guaranteed not to attempt a transmission during these steps by the definition of σ'_γ.

Since the values of $\rho_{i,t}$ and $\rho^*_{i,t}$ for $t \in \sigma'_\gamma$ and $t \notin \sigma'_\gamma$ are independent, we can conclude that all of the above constraints on the ρ-values are satisfied with probability

$$\Omega((2NS)!^{-\alpha})\left(1 - O\left(N2^{-c\sqrt{S}} + N^2S^{1-\alpha} + \frac{N}{U}\right)\right).$$

Of course, we still must show what values of $\rho_{i,t}$ and $\rho^*_{i,t}$ that satisfy these constraints actually produce the desired sequence of events when we interleave steps in σ'_γ with steps not in σ'_γ in the correct order. Once this is accomplished, we are done since we will have shown that with probability exceeding

$$\Omega((2NS)!^{-\alpha})\left(1 - O\left(N2^{-c\sqrt{S}} + N^2S^{1-\alpha} + \frac{N}{U}\right)\right),$$

the system behaves as claimed.

The proof that the real system behaves well if the $\rho_{i,t}$ and $\rho^*_{i,t}$ values satisfy the proceeding constraints proceeds by induction over t. The base case $t = 2$ was already established. We then consider what happens at some time $t \geq 3$, assuming that previous moves in the real system were essentially identical to moves of the corresponding steps of the isolated systems. If $t \in \sigma'_\gamma$, then we can be assured that $b_N \leq 1$ before the interval I_i that contains t and thus that S_N broadcasts and that stations with big backoff counters do not broadcast. Hence stations with small backoff counters are blocked and their behavior continues to agree with the assumptions that were used to define σ'_γ. Hence the definition of σ'_γ as describing when stations with small backoff counters attempt to broadcast remains valid. This is precisely what we want to have happen. For $t \notin \sigma'_\gamma$, the system behaves exactly as it does in the scenario when we ignored σ'_γ because the activity in σ'_γ has no effect on any of the large backoff counters and because stations with small backoff counters are guaranteed not to try anything by the definition of σ'_γ. The only possible difference is that b_N could be lowered from 1 to 0 by including some steps of σ'_γ. The only effect of making $b_N = 0$, however, is to start S_N broadcasting sooner. By these constraints, we know that S_N will be the next station to transmit anyway, so starting off sooner only increases the number of successful transmissions without otherwise changing that state of the system. Hence the behavior of the combined system is virtually identical to its behavior during σ'_γ and outside σ'_γ when considered in isolation, provided that the constraints on the ρ values are satisfied.

The formal justification that events go well with the claimed probability is now complete. All that remains is to bound the possible increase in the potential function should any of these conditions fail. Note that no matter what the reason for failure, all the conditions held in the previous step by assumption. Hence the most we could have added to the potential function because of the counters is $O(NU^{(\alpha+1/2)/(\alpha+1)})$. Similarly, the most we could add (net) because of the queues is $O(U^{1/2}\log U + NU^{1/(\alpha+1)}\log U)$. Hence, the worst increase we could suffer is $O(NU^{(\alpha+1/2)/(\alpha+1)})$.

Putting everything together, we find that there is a tree of descendent states with depth at most U for which the expected decrease in potential is at least

$$\Omega(B^{-2\alpha}(1-\lambda)2^{-N}(2NS)!^{-\alpha})\left((1-\lambda)U - O\left(NU^{\frac{\alpha+1/2}{\alpha+1}}\right)\right) - O\left(NU^{\frac{\alpha+1/2}{\alpha+1}}\right)$$
$$\geq \Omega(B^{-2\alpha}(1-\lambda)^2U2^{-N}(2NS)!^{-\alpha}) - O\left(NU^{\frac{\alpha+1/2}{\alpha+1}}\right).$$

By selecting

$$U = \left(\frac{c_U 2^N N B^{2\alpha} (2NS)!^{\alpha}}{(1-\lambda)^2} \right)^{2\alpha+2}$$

for some constant c_U independent of N and λ, we get $\mathrm{Ex}[\Delta POT] \leq -\delta$.

To complete the proof of Theorem 3.1, we need only observe that $\mathrm{Ex}[(\Delta POT)^2]$ does not cause any problems since all the ΔPOTs are of order U, and if we choose $V = \Omega(U^3)$, we are done.

Remark. Let us just point out that there is no problem in determining our constants. The reason is that our conditions can be summarized as follows:

$$\begin{aligned} M &\geq f_1(\lambda, N), \\ B &\geq f_2(\lambda, N, M), \\ S &\geq f_3(\lambda, N), \\ U &\geq f_4(\lambda, N, B, S), \\ V &\geq f_5(U, B). \end{aligned}$$

Here f_i are the explicit functions given in the proof.

4. Lower bounds on Ex[L_{ave}]. The analysis presented in §3 reveals that $\mathrm{Ex}[L_{\text{ave}}]$ is at most $P\left((1-\lambda)^{-1}\right)2^{Q(N)}$, where P and Q are polynomial functions. We do not know whether or not the dependence on N can be made polynomial. The main difficulty in proving a polynomial upper bound in N by extending our methods lies in analyzing the probability that a station will grab control of the channel and empty its queue.

We can prove nontrivial lower bounds on $\mathrm{Ex}[L_{\text{ave}}]$, however. In particular, in this section, we show that for a wide range of backoff functions the expected number of nonempty queues over time is linear in the number of stations. For many backoff functions, this fact will imply that $\mathrm{Ex}[L_{\text{ave}}]$ grows superlinearly in N.

We will assume that all stations have probability $\frac{\lambda}{N}$ of getting a message at each timeslot. This is a reasonable assumption since if the arriving messages are very unevenly distributed among the stations, the system would in reality be a system with fewer than N stations. On the other hand, small deviations from this assumption can be handled.

Let us start by giving an outline of the ideas of this section. The basic tool will be to establish a connection between our finite model and the infinite model briefly discussed in §2.3. We will not prove that the models behave in the same way but rather that the proofs of instability in the infinite model extend to give lower bounds in our finite model. Before we can make this precise, however, we need a more formal definition of the infinite model.

In the infinite model, there is a countably infinite number of stations and no station ever gets two messages. The total number of messages that arrive to the system at a given time t is assumed to be Poisson distributed with mean λ. Each station behaves exactly in the same way as in the finite model. Since there are never two messages in any station, it is convenient to talk of the system as if each message had its own backoff counter. We next review some results for the infinite model.

Assume for the moment that the system is continuously externally jammed (i.e., that no transmissions are successful) and that a message arrives at time 1. Define $h(x)$ to be the probability that an attempt is made to transmit this message at time x. For example, $\mathrm{Ex}[h(x)] = \Theta(x^{-\alpha/(\alpha+1)})$ if $f(b) = (b+1)^{-\alpha}$. Let

$$H(\lambda) = \sum_{t=1}^{\infty} \left(1 + \lambda \sum_{x=1}^{t} h(x) \right) e^{-\lambda \sum_{x=1}^{t} h(x)}.$$

We will say that a backoff function has property H_λ if there exist λ' such that $\lambda > \lambda'$ and $H(\lambda')$ is finite. Using this notation, we have the following theorem due to Kelly [5].

THEOREM (Kelly [5]). *Consider the infinite model and any backoff function satisfying property H_λ. Then the expected number of successful transmissions up to time T is bounded by a constant independent of T when the arrival rate is λ.*

It is not too difficult to check that essentially any function which grows slower than any exponential function has property H_λ for any $\lambda > 0$. In particular, this is true for any polynomial backoff function. The previous theorem does not apply to $f(b) = 2^{-b}$ for $\lambda < \ln 2$, however. In this case, we need to rely on the following almost equally strong theorem.

THEOREM (Aldous [1]). *Consider the infinite model and $f(b) = 2^{-b}$, $\lambda \leq \ln 2$. Then for any $a > 1 - \frac{\lambda}{\ln 2}$, the expected number of successful transmissions up to time T is $o(T^a)$.*

It will help to provide a brief outline for the proofs of these theorems. Let the *mass* $m(t)$ of the system at time t be defined by $m(t) = \sum_i f(b_i)$. A fact which underlies most of the analysis is that the probability of a successful transmission is bounded above by $(\lambda + m(t))e^{1-m(t)}$. An easy calculation shows that this is true in both the finite and infinite models.

Using this fact, we can now give the idea behind the proofs. Within constant expected time, due to many messages arriving within a short time interval, the mass will exceed K for some large given constant K. Once this happens, the probability of success is small and the messages that arrive to the system are not successfully transmitted, which implies that the mass increases even more, and so on.

To get a connection between the infinite and finite models, we will forget any message that is not an *active* message (i.e., first in its queue).

When we disregard messages which are not active, the finite model behaves in a similar way to the infinite model as long as the number of stations with empty queues remains at least cN. The only differences are

(1) after a successful transmission, an additional new active message might appear in the finite model, due to the fact that the station in question has a queue of length 2 or more, and

(2) the distribution of the number of arriving active messages is not constant over time in the finite model (it depends on the number of stations with empty queues) and is not Poisson (it is a sum of binomials instead).

Although these differences are substantial, for the most part they tend to just make the behavior of the finite system worse than its infinite counterpart as long as $x_e(t) = \Omega(N)$, where $x_e(t)$ is the number of empty queues in the finite system at time t. In particular, we can establish the following key property.

LEMMA 4.1. *Suppose that $x_e(t) \geq cN$ in an N-station system with arrival rates $\lambda_i \geq \frac{\lambda}{N}$. Then there a constants $K_{\lambda,c}$ and $d_{\lambda,c}$ such that within expected time $K_{\lambda,c}$, there is a point in time t_0 such that $m(t_0) \geq d_{\lambda,c}$ such that for every t,*

$$m(t_0 + t) \geq \begin{cases} d_{\lambda,c} \log t & \textit{for } f(b) = 2^{-b}, \\ d_{\lambda,c} t^{\frac{1}{\alpha+1}} & \textit{for } f(b) = b^{-\alpha}, \end{cases}$$

or $x_e(t_0 + s) \leq cN$ for some $s \in [1, t]$

Proof sketch. Let us first take care of the case $f(b) = 2^{-b}$. Lemma 4.1 describes the mechanism that Aldous [1] uses in his proof. We will not repeat the proof here but just describe how to take care of the differences. Aldous uses two key lemmas, one which states that there are not too many successful transmissions (Lemma 3) and one which states that there are many new arrivals (Lemma 4). In our situation, the proof of his Lemma 3 goes through virtually without change. To prove the equivalent of Lemma 4, one needs to take care of the differences described above. Difference (1) only helps us since it provides extra arrivals. To take care of difference (2), observe that what is needed is an estimate that much fewer messages arrive than expected. By our assumption on the number of empty queues and the arrival rates, the expected number of arriving messages is high and the probability of getting only a fraction which is $\frac{7}{9}$ of the expected value is exponentially small. We conclude that the bounds also hold in our case.

The case $f(b) = b^{-\alpha}$ is easier and can be taken care of in two ways, either by imitating Aldous's proof or by extending Kelly's proof to show that the expected number of transmissions before $x_e(t) \leq cN$ is a constant. The differences in the two models are taken care of in a similar way. □

Define

$$X_e(T) = \frac{1}{T}\sum_{t=1}^{T} x_e(t)$$

to be the average number of empty queues over time. We use Lemma 4.1 to bound $\mathrm{Ex}[X_e(T)]$.

THEOREM 4.2. *Let $f(b) = b^{-\alpha}$ or $f(b) = 2^{-b}$. Then for any $c > 0$, $E(X_e(T)) \leq cN + o(N)$ for $T \geq d_{c,\lambda}N$.*

Proof. Let $c' = \frac{c}{2}$. We know by Lemma 4.1 that if $x_e(t) \geq c'N$, then within constant expected time the system will reach a state with $m(t) \geq -10\log c\lambda$ and remain this way until $x_e(t) \leq c'N$. Once the mass is this large, successes happen with probability $\leq \frac{c\lambda}{2}$. Since a message arrives in an empty queue with probability at least $c\lambda$, the number of empty queues will constitute a biased random walk. Furthermore, the probability of not going into a high-mass situation within time i is bounded by 2^{-ki} for some $k > 0$. This implies that if $x_e(0) \leq c'N$, then for any $t > 0$, $\Pr[x_e(t) \geq c'N + i] \leq 2^{-ki}$ again with $k > 0$. If, on the other hand, $x_e(0) > c'N$, the probability that $x_e(t)$ remains greater than $c'N$ for time dN for some large d is less than $2e^{-d'N}$. This follows since with probability $e^{-d'N}$, the system will enter the state prescribed by Lemma 4.1 before time $\frac{d}{2}N$ and the probability that the biased random walks stays above $x_e(t) \geq c'N$ is $e^{-d'N}$. From then on, the previous case applies. In either case, we know that for any $t \geq dN$, the probability that $x_e(t) \geq cN$ is $\leq e^{-d'N}$, and this proves the theorem. □

Having established that, on the average, we have a linear number of nonempty queues, we now look at the length of these queues.

LEMMA 4.3. *If the system is stable and $f(b) = 2^{-b}$ or $f(b) = b^{-\alpha}$, then for sufficiently large N, a fraction c_λ of the time $x_e(t) \leq \frac{2N}{3}$ and $m(t) \leq R_\lambda$.*

Proof. To have a stable system with total arrival rate λ, the probability of success has to be $\geq \frac{\lambda}{2}$ at least a fraction $\frac{\lambda}{2}$ of the time. This implies that $m(t) \leq R$ at least $\frac{\lambda}{2}$ of the time, where R is a constant depending on λ. Furthermore, whenever $x_e(t) \geq \frac{2N}{3}$, by Lemma 4.1 within constant expected time $m(t)$ will exceed R and stay that way for time at least cN. Thus the fraction of the time for which $m(t) < R$ and $x_e(t) \geq \frac{2N}{3}$ is bounded by $\frac{c}{N}$, and the lemma follows. □

LEMMA 4.4. *If $f(b) = (b+1)^{-\alpha}$, $\alpha > 1$, then a fraction c_λ of the time there are $\Omega(N^{(\alpha+1)/\alpha})$ messages in the queues.*

Proof. We know by Theorem 3.1 that the system is stable and thus every state S of the system has a probability $\Pr(S)$ associated which is the relative frequency with which the system is in state S. We know that

$$\sum_{S, m(S)\leq R, x_e(S)\leq \frac{2N}{3}} \Pr(S) \geq c_\lambda.$$

For any state in the above sum there are $\Omega(N)$ queues whose backoff counters are at least $d_\lambda N^{\frac{1}{\alpha}}$. Define station i to be *unusual* if $\lambda b_i^{\alpha+1}/4N(\alpha+1) > q_i$ and $b_i \geq d_\lambda N^{\frac{1}{\alpha}}$, where d_λ

is a constant to be determined. Say that any point in time is *unusual* if at least $\sqrt{N}$ of the stations are unusual. Using Lemma 3.4, it follows that the fraction of unusual points in time is exponentially small. Thus the states S which are not unusual and have $m(S) \le R$ and $x_e(S) \le \frac{2N}{3}$ have total probability $\ge c'$ for $N \ge N_\lambda$. But any such state has $\Omega(N)$ stations each with $\Omega(N^{1/\alpha})$ long queues, and the lemma follows. □

Lemma 4.4 immediately implies the following lower bound on $\mathrm{Ex}[L_{\mathrm{ave}}]$.

THEOREM 4.5. *If* $f(b) = (b+1)^{-\alpha}$, *then* $\mathrm{Ex}[L_{\mathrm{ave}}] \ge \Omega(N^{\frac{\alpha+1}{\alpha}})$.

For quadratic backoff, this means that $\mathrm{Ex}[L_{\mathrm{ave}}] \ge \Omega(N^{\frac{3}{2}})$.

5. Instability of exponential backoff. In this section, we prove instability results for binary exponential backoff. We will prove two results; one which is exact and the other asymptotic. Let us start by stating the exact result.

THEOREM 5.1. *Suppose binary exponential backoff is used and the arrival rate at every station is* $\frac{\lambda}{N}$, *where* $\lambda > \lambda_0 + \frac{1}{4N-2}$ *and* $\lambda_0 \approx 0.567$ *is the solution to* $\lambda_0 = e^{-\lambda_0}$. *Then the system is unstable.*

To prove the result, we use the potential function

$$POT = C\sum_{i=1}^{N} q_i + \sum_{i=1}^{N} 2^{b_i} - N.$$

The best choice for C will turn out to be $2N-1$.

For any state in the system, we will show that the potential function is expected to increase by at least a fixed amount (independent of the state) during the subsequent transition. This will enable us to prove Theorem 5.1.

The proof requires the use of the following simple lemma.

LEMMA 5.2. *If* $0 \le \epsilon_i \le 1$ *for* $1 \le i \le m$, *then*

$$\prod_{i=1}^{m}(1+\epsilon_i) \ge 1 + \sum_{i=1}^{m}\epsilon_i$$

and

$$\prod_{i=1}^{m}(1+\epsilon_i) \ge 2\sum_{i=1}^{m}\epsilon_i.$$

Proof. The first inequality is obvious from expansion of the product and the nonnegativity of the ϵ_i's. The second inequality follows from the observation that

$$\prod_{i=1}^{m}(1+\epsilon_i) - \sum_{i=1}^{m}2\epsilon_i \ge \prod_{i=1}^{m}(1-\epsilon_i) \ge 0$$

since $\epsilon_i \le 1$ for $1 \le i \le m$. □

In the proof, we let M denote the number of stations with a nonzero backoff counter. Without loss of generality, we can assume that $b_1, \ldots, b_M \ne 0$ and $b_{M+1}, \ldots, b_N = 0$, where $0 \le M \le N$. Note that if $b_i \ne 0$, then $q_i \ne 0$ since there must be some message that failed in its most recent attempt to transmit. In addition, the queues in all but one of the stations $M+1, \ldots, N$ must be zero. This is because any station with $b_i = 0$ and $q_i \ne 0$ must have successfully transmitted during the last step. Hence we divide our analysis into two cases, depending on whether or not $q_{M+1} = 0$.

Case I. $b_1, \ldots, b_M \ne 0; b_{M+1}, \cdots, b_N = 0; q_{M+1}, \ldots, q_N = 0; 0 \le M \le N$.

We start with some additional notation. As in §3, we let

$$p_i = \begin{cases} 2^{-b_i} & \text{for } 1 \le i \le M, \\ \lambda_i & \text{for } M+1 \le i \le N \end{cases}$$

be the probability that the ith station attempts to transmit, where $\lambda_i = \frac{\lambda}{N}$ is the probability that a new message arrives at the ith station. Let

$$T = \prod_{i=1}^{N}(1 - p_i)$$

be the probability that none of the N stations attempts to transmit a message. In addition, the probability that none of $\{1, \ldots, i-1, i+1, \ldots, N\}$ try to transmit is $T/(1-p_i)$.

We also define

$$\epsilon_i = \frac{p_i}{1-p_i}, \quad R = \prod_{i=1}^{N}(1+\epsilon_i), \quad \text{and} \quad S = \sum_{i=1}^{N}\epsilon_i.$$

Note that $1 + \epsilon_i$ is $1/(1-p_i)$, so $RT = 1$. Also note that $0 \le \epsilon_i \le 1$ for $1 \le i \le N$ since $b_i \ge 1$ for $1 \le i \le M$ and $\lambda_i = \frac{\lambda}{N} \le \frac{1}{2}$ for $N \ge 2$.

We let Q_i^+, Q_i^-, B_i^+, and B_i^- denote the same quantities as in §3, and since expectations sum, we have

$$\mathrm{Ex}[\Delta POT] = \sum_{i=1}^{N} Q_i^+ - \sum_{i=1}^{N} Q_i^- + \sum_{i=1}^{N} B_i^+ - \sum_{i=1}^{N} B_i^-.$$

It is easily seen that $Q_i^+ = C\lambda_i$ for $1 \le i \le N$. Since the ith station transmits successfully with probability $T\epsilon_i$, we can also easily conclude that $Q_i^- = CT\epsilon_i$ for $1 \le i \le N$. Since the ith station crashes with probability $(1 - T/(1-p_i))p_i$, the value of B_i^+ is

$$\left(1 - \frac{T}{1-2^{-b_i}}\right)2^{-b_i} \cdot 2^{b_i} = 1 - T(1+\epsilon_i)$$

for $1 \le i \le M$, and

$$\left(1 - \frac{T}{1-\lambda_i}\right)\lambda_i = \lambda_i - T\epsilon_i$$

for $M + 1 \le i \le N$. Finally, we note that

$$B_i^- = \frac{T}{1-2^{-b_i}}2^{-b_i}(2^{b_i} - 1) = T$$

for $1 \le i \le M$ and that $B_i^- = 0$ otherwise.

Summing these values over $1 \le i \le N$, we find that

$$\begin{aligned}\mathrm{Ex}[\Delta POT] &= C\lambda - CTS + M - MT - TS + \sum_{i=M+1}^{N}\lambda_i - MT \\ &= C\lambda + M + \sum_{i=M+1}^{N}\lambda_i - T[(C+1)S + 2M].\end{aligned}$$

Hence $\mathrm{Ex}[\Delta POT] \ge \delta$ if and only if

$$R\left(C\lambda - \delta + M + \sum_{i=M+1}^{N}\lambda_i\right) > (C+1)S + 2M.$$

Substituting $C = 2N - 1$ and $\lambda \geq \frac{1}{2} + \frac{1}{2C} + \frac{\delta}{C}$, we need only check that

$$R(N + M) \geq 2NS + 2M$$

in order to verify that $\text{Ex}[\Delta POT] \geq \delta$ for any $\delta > 0$. This inequality easily follows from Lemma 5.2 since if $S \geq 1$, we use the facts that $R \geq 2S$ and $R \geq 2$, and if $S \leq 1$, we use the facts that $R \geq 1 + S$ and $N \geq M$.

Case II. $b_1, \ldots, b_M \neq 0$; $b_{M+1}, \ldots, b_N = 0$; $q_{M+1} \neq 0$; $q_{M+2}, \ldots, q_N = 0$; $0 \leq M < N$.

In this case, we are guaranteed that station $M + 1$ will attempt a transmission. The probability that it is successful is

$$W = \prod_{i=1}^{M}(1 - 2^{-b_i}) \prod_{i=M+2}^{N} (1 - \lambda_i).$$

The analysis for the Q_i's and B_i's is similar to Case I. In particular, $Q_i^+ = C\lambda_i$,

$$Q_i^- = \begin{cases} CW & \text{for } i = M + 1, \\ 0 & \text{otherwise}, \end{cases}$$

$$B_i^+ = \begin{cases} 1 & \text{for } 1 \leq i \leq M, \\ 1 - W & \text{for } i = M + 1, \\ \lambda_i & \text{for } M + 2 \leq i \leq N, \end{cases}$$

and $B_i^- = 0$ for $1 \leq i \leq N$. Summing these values, we find that

$$\text{Ex}[\Delta POT] = C\lambda - CW + M + 1 - W + \sum_{i=M+2}^{N} \lambda_i.$$

Thus $\text{Ex}[\Delta POT] \geq \delta$ if and only if

$$C\lambda + M + 1 + \sum_{i=M+2}^{N} \lambda_i \geq (C + 1)W + \delta.$$

This is just a calculation and we defer it to the appendix.

Having established that we have an expected increase in potential each step let us see how we can use that to establish Theorem 5.1. There are general conditions under which increase in potential implies instability (see, for instance, [14]). However, to verify these conditions require a fair amount of additional work and we believe that a direct proof is more illuminating.

Let us first prove that the expected waiting time is infinite over time.

LEMMA 5.3. *At time T the expected waiting time for a newly arrived message is δT for some positive constant δ.*

A message that arrives at S_i has expected waiting time at least $q_i + 2^{b_i}$, the reason being that the expected time before the first message in the queue is sent is 2^{b_i} and then at most one message can be sent per time step. The probability of an arriving message arriving at S_i is at least $\frac{\lambda}{N}$ if we assume that the expected number of arrivals per time step is less than 1 (otherwise the theorem is trivial). Thus the expected waiting time is at least

$$\frac{\lambda}{N} \sum_{i=1}^{N}(q_i + 2^{b_i}) \geq \frac{1}{4N^2} POT.$$

Since the expected value of the potential is $\Omega(T)$ we are done. □

Thus we have established that the expected waiting time and hence the expected queue size gets arbitrarily large as time goes by. Let us proceed to prove that the recurrence time is infinite.

Suppose we start at state where all queues are empty. We want to prove that the expected time to return to this state is infinite. We know that in time T the expected potential is δT. As an extension of this, we first establish that for some constants c and d, it is true that with probability at least c the potential is at least dT. The essential lemma towards establishing this is as follows.

LEMMA 5.4. *For any $b \geq \lceil \log T \rceil$, the probability that the ith backoff counter has reached b in time T is bounded by $2^{-(b-\lceil \log_2 T \rceil)}$.*

Proof. There must have been $b - \lceil \log T \rceil$ increases in the backoff counter after it reached $\lceil \log T \rceil$. There are

$$\binom{T}{b - \lceil \log T \rceil} \leq \frac{T^{b-\lceil \log T \rceil}}{(b - \lceil \log T \rceil)!}$$

possible ways to choose the time slots where these increases could happen. For any fixed choice of these time slots, the probability that the backoff counter would increase at these time slots is at most

$$\prod_{i=\lceil \log T \rceil}^{b-1} 2^{-i} \leq T^{-(b-\lceil \log T \rceil)} 2^{-(b-\lceil \log_2 T \rceil)}.$$

Multiplying out the lemma follows. □

Next we prove the following result.

LEMMA 5.5. *With probability at least c the potential at time T is at least dT.*

Proof. Let S denote a state of the system and consider the following claim.

CLAIM. *There is a constant D such that*

$$\sum_{S, POT(S) \geq DT} \Pr[S] POT(S) \leq \frac{\delta}{2} T.$$

Before establishing the claim, let us see how the lemma follows. Since $E(POT(S)) \geq \delta T$, the claim implies that

$$\sum_{S, POT(S) \leq DT} \Pr[S] POT(S) \geq \frac{\delta}{2} T.$$

But this clearly implies that $\Pr[POT(S) \geq \frac{\delta T}{4}] \geq \frac{\delta}{4D}$.

Thus we only have to establish the claim. Suppose $D > 4N^2$. Since no queue can be longer than T, the contribution from the queues to the potential is bounded by $2N^2 T$. Thus for the potential to exceed DT, it is necessary that $2^{b_i} \geq \frac{DT}{2N}$ for some i. Using that in this case the contribution to the potential from the queue lengths is bounded by the contribution from the backoff counters, we get the estimate

$$\sum_{i=1}^{N} \sum_{S, b_i(S) \geq b_j(S), j \neq i, POT(S) \geq DT} \Pr[S] POT(S) \leq N \sum_{b=\lceil \log \frac{DT}{2N} \rceil}^{\infty} 2N 2^b \Pr[b_1 = b]$$

$$\leq 2N^2 \sum_{b=\lceil \log \frac{DT}{2N} \rceil}^{\infty} 2^{b-\binom{b-\lceil \log T \rceil}{2}} \leq 4TN^2 \sum_{i=\lceil \log \frac{D}{2N} \rceil}^{\infty} 2^{i-\binom{i}{2}} \leq \frac{\delta}{2} T$$

for $D > D_\delta$. □

Now we are ready for the final part of the proof of Theorem 5.1. From Lemma 5.5 it follows that with probability at least $\frac{c}{T}$ the potential reaches at least dT before it returns to 0. Observe that the expected time to return from potential P to potential 0 is at least $\frac{P}{2N}$. This follows from looking at the largest backoff counter or the longest queue. Using this we get

$$\text{Ex(return time)} \geq \frac{1}{2N} \text{ Ex (Maximum potential before return)}$$

$$= \frac{1}{2N} \sum_{i=1}^{\infty} i \Pr[\text{Max pot } = i] = \frac{1}{2N} \sum_{i=1}^{\infty} \Pr[\text{Max pot } \geq i] \geq \sum_{i=1}^{\infty} \frac{cd}{2Ni}.$$

However this last sum diverges and we have proved Theorem 5.1. □

Using the results from §4, we strengthen Theorem 5.1 slightly in an asymptotic sense.

THEOREM 5.6. *Suppose binary exponential backoff is used and the arrival rate at every station is* $\frac{\lambda}{N}$, *where* $\lambda \geq c$, *where* $c > \frac{1}{2}$. *Then the expected recurrence time is infinite for* $N > N_c$.

Proof. Since the proof of Theorem 5.6 is almost identical to that of Theorem 5.1, we will only point out the modifications needed.

We will be working with the same potential function and will again show that the potential is expected to increase. However, in this case, we are sometimes forced to consider more than one step of the system to obtain the desired increase. We use the same cases as in the proof of Theorem 5.1. Observe first that in Case I, we only needed $\lambda \geq \frac{1}{2} + \frac{1}{2C} + \frac{\delta}{C}$ to obtain the expected increase. Since $C > N$, this bound is $< c$ for $N > N_c$ and $\delta < 1$.

To handle Case II will require some work. We get two subcases depending on the number of stations with empty queues. Lemma 5.7 takes care of the first case.

LEMMA 5.7. *For* $c > \frac{1}{2}$, *there is a constant* d_c, $0 < d_c < 1$, *such that if* $\leq d_c N$ *queues are nonempty, the expected increase in potential is* $> \delta$.

Proof. We know by the analysis in Case II of Theorem 5.1 that the expected decrease if M stations have empty queues is given by

$$g(M) = 2N\lambda + (M+1)\left(1 - \frac{\lambda}{N}\right) - 2N\left(1 - \frac{\lambda}{N}\right)^{N-M-1}.$$

To prove that $g(M)$ is positive in the claimed interval, we proceed as before by establishing that $g(N) > 0$, $g(d_c N) > 0$, and $(\delta^2/\delta M^2) g(M) < 0$. Since the first and last condition was taken care of in the proof of Theorem 5.1, we need only to establish the second condition. It is easy to see that $g(d_c N) > 0$ for sufficiently large N and $\lambda = c$ if and only if

$$\tilde{g}(d_c, c) = 2c + d_c - 2e^{-c(1-d_c)} > 0.$$

But since $\tilde{g}(d_c, c)$ is a continuous function in c and d_c and $\tilde{g}(1, c) > 0$, the lemma follows. □

To take care of the case of many empty queues, we have the following result.

LEMMA 5.8. *Let* $\lambda \geq c > \frac{1}{2}$; *then there is a constant* K_c *such that if* $x_e(t) \geq d_c N$, *then the expected change in potential over the next* K_c *steps is* $\geq \delta$.

Proof. By the previous analysis, the expected change in potential in a step is bounded from below by $-hN$ for some constant $h > 0$. Let $K' = \min(\frac{K}{2}, \frac{K}{32h})$, where K is a constant such that if $x_e(t) \geq (1 - d_c)N$, then the probability that $m(t + t_0) \geq 5$ for $t_0 \in [K', K]$ is $\geq 1 - \frac{K'}{K}$. Such a constant exist for sufficiently large N by Lemma 4.1 applied with $c = (1 - d_c)/2$. Consider the system over the next K steps. Observe that if $m(t + t_0) > 5$, then the expected increase in potential is at least $(c - \frac{1}{4})2N$. This follows since the probability of a successful transmission is $\leq \frac{1}{4}$ and the the contribution from the backoff counters is expected to increase. Let Δ_i be the expected change in potential in potential at time $t+i$. Then

$$\begin{aligned}\sum_{i=1}^{K} \Delta_i &= \sum_{i=1}^{K'} \Delta_i + \sum_{i=K'+1}^{K} \Delta_i \\ &\geq -hNK' + \left(1 - \frac{K'}{K}\right)(K-K')\left(c - \frac{1}{4}\right)2N - hN\frac{K'}{K}(K-K') \\ &\geq NK\left(\frac{1}{8} - \frac{2}{32}\right) \geq \frac{NK}{16}.\end{aligned}$$

This concludes the proof of Lemma 5.8. □

Using Lemmas 5.7 and 5.8, we know that the expected increase of potential over T steps is $\geq \delta T$ for some constant δ. We go from large expected potential to infinite recurrence time as was done in Theorem 5.1 and this completes the proof of Theorem 5.6. □

6. Instability of linear and sublinear backoff. In this section, we study linear and sublinear backoff, and the goal of the current section is to prove the following theorem.

THEOREM 6.1. *If $f(b) = (1+b)^{-\alpha}, 0 < \alpha \leq 1$, is used as a backoff function, then for any $\lambda > 0$ and $N > N_\lambda$, the system is unstable.*

As before, we derive our result by using a potential function. In this case, we use

$$POT = N^{\frac{3}{2}} \sum_{i=1}^{N} q_i - \sum_{i=1}^{N} (b_i + 1)^{\alpha+1}.$$

We will first analyze the expected change in one or two steps. We do this by establishing a series of lemmas, and we start by giving some facts which are needed in several places. Let s be the number of nonempty queues. Let P_{suc} denote the probability of success. Then

$$\begin{aligned}P_{\text{suc}} = &\left(1 - \frac{\lambda}{N}\right)^{N-s} \sum_{i=1}^{s} (1+b_i)^{-\alpha} \prod_{j \leq s, i \neq j} \left(1 - (b_j+1)^{-\alpha}\right) \\ &+ \left(1 - \frac{\lambda}{N}\right)^{N-s-1} \frac{(N-s)\lambda}{N} \prod_{j=1}^{s} \left(1 - (b_j+1)^{-\alpha}\right).\end{aligned}$$

As in previous sections, we need the expected change in the two components of POT. Observe that B^+ corresponds to the increase in POT and hence the decrease of $\sum_{i=1}^{N} b_i^2$. By straightforward analysis, we have

$$Q^+ = \lambda N^{\frac{3}{2}},$$

$$Q^- = P_{\text{suc}} N^{\frac{3}{2}},$$

$$B^+ = \left(1 - \frac{\lambda}{N}\right)^{N-s} \sum_{i=1}^{s} (1+b_i)^{-\alpha}((b_i+1)^{\alpha+1} - 1) \prod_{j \leq s, i \neq j} (1 - (1+b_j)^{-\alpha}),$$

$$\begin{aligned}B^- = &\sum_{i=1}^{s} (1+b_i)^{-\alpha} \left(1 - \left(1 - \frac{\lambda}{N}\right)^{N-s} \prod_{j \leq s, i \neq j} (1 - (1+b_j)^{-\alpha})\right)((b_i+2)^{\alpha+1} - (b_i+1)^{\alpha+1}) \\ &+ \frac{(N-s)\lambda}{N}\left(1 - \left(1 - \frac{\lambda}{N}\right)^{N-s-1} \prod_{j \leq s} (1 - (1+b_j)^{-\alpha})\right).\end{aligned}$$

B^- will not play any significant role in the analysis, and the reason for this is the following fact.

Fact 1. $B^- \leq 4N$.

This follows from

$$B^- = \sum_{i=1}^{s}(1+b_i)^{-\alpha}\left(1-\left(1-\frac{\lambda}{N}\right)^{N-s} ! \prod_{j\leq s, i\neq j}\left(1-(1+b_j)^{-\alpha}\right)\right)\left((b_i+2)^{\alpha+1}-(b_i+1)^{\alpha+1}\right)$$
$$+\frac{(N-s)\lambda}{N}\left(1-\left(1-\frac{\lambda}{N}\right)^{N-s-1}\prod_{j\leq s}\left(1-(1+b_j)^{-\alpha}\right)\right)$$
$$\leq \sum_{i=1}^{s}(1+b_i)^{-\alpha}\left((b_i+2)^{\alpha+1}-(b_i+1)^{\alpha+1}\right)+\frac{(N-s)\lambda}{N} \leq 4s+N-s \leq 4N$$

using $((x+1)^{\alpha+1}-x^{\alpha+1})/x^{\alpha} \leq (\alpha+1)(x+1)^{\alpha}/x^{\alpha} \leq (\alpha+1)2^{\alpha} \leq 4$. The first inequality follows from taking the maximal value of the derivative and the last follows from $\alpha \leq 1$.

Let us next take care of the easy case of estimating the change in potential.

LEMMA 6.2. *If $P_{\text{suc}} < \frac{\lambda}{2}$, then $\Delta POT \geq \frac{\lambda}{3}N^{3/2}$ for $N > N_\lambda$.*

Proof. We have

$$\Delta POT \geq Q^+ - Q^- - B^- \geq \lambda N^{\frac{3}{2}} - \frac{\lambda}{2}N^{\frac{3}{2}} - 4N \geq \frac{\lambda}{3}N^{\frac{3}{2}}$$

for $N > N_\lambda$. □

In the future, let c_λ be an arbitrary constant whose values depends on λ. We will assume that the value of c_λ may change from line to line, and thus $2c_\lambda \leq c_\lambda$ is a valid inequality. We are now considering $P_{\text{suc}} \geq \frac{\lambda}{2}$, and observe that this implies that $\sum_{i=1}^{s}(b_i+1)^{-\alpha}+(N-s)\frac{\lambda}{N} \leq K_\lambda$, where K_λ is a constant close to $-\log\lambda$. Next we have the following result.

LEMMA 6.3. *If $b_i > 0$ for $1 \leq i \leq s$ and $P_{\text{suc}} \geq \frac{\lambda}{2}$, then $\Delta POT \geq c_\lambda s^2 - N^{\frac{3}{2}} - 4N$.*

Proof. The main contribution to the increase of the potential this time will come from B^+.

$$B^+ = \left(1-\frac{\lambda}{N}\right)^{N-s}\sum_{i=1}^{s}\frac{b_i^{1+\alpha}}{(1+b_i)^{\alpha}}\prod_{j\leq s, i\neq j}\left(1-(1+b_j)^{-\alpha}\right)$$
$$\geq c_\lambda\sum_{i=1}^{s}\frac{b_i^{2\alpha}}{(1+b_i)^{\alpha}} \geq c_\lambda s^2.$$

The first inequality comes from $\sum_{i=1}^{s}(b_i+1)^{-\alpha}+(N-s)\frac{\lambda}{N} \leq K_\lambda$ and the second inequality follows from Hölder's inequality since

$$\frac{s}{2} \leq \sum_{i=1}^{s}\frac{b_i^{\alpha}}{(b_i+1)^{\alpha}} \leq \sum_{i=1}^{s}\frac{b_i^{\alpha}}{(b_i+1)^{\frac{\alpha}{2}}}\frac{1}{(b_i+1)^{\frac{\alpha}{2}}} \leq \left(\sum_{i=1}^{s}\frac{b_i^{2\alpha}}{(1+b_i)^{\alpha}}\right)^{\frac{1}{2}}\left(\sum_{i=1}^{s}\frac{1}{(1+b_i)^{\alpha}}\right)^{\frac{1}{2}}$$

and again using that the last sum is bounded. The lemma now follows since $Q^- \leq N^{\frac{3}{2}}$ and $B^- \leq 4N$. □

Finally, we must take care of the case when $b_1 = 0$ and $P_{\text{suc}} \geq \frac{\lambda}{2}$.

LEMMA 6.4. *If $b_1 = 0$ and $b_i > 0$ for $2 \leq i \leq s$ and $P_{\text{suc}} \geq \frac{\lambda}{2}$, then $\Delta POT_{\text{two steps}} \geq c_\lambda s^2 - 2N^{\frac{3}{2}} - 8N$.*

Proof. The increase will come from B^+ in the case when a collision appears at step 1. The probability of collision at step 1 is

$$1-\left(1-\frac{\lambda}{N}\right)^{N-s}\prod_{i=2}^{s}\left(1-(1+b_i)^{-\alpha}\right)$$

$$\geq 1-\prod_{i=2}^{s}\left(1-(1+b_i)^{-\alpha}\right)\geq c_\lambda\sum_{i=1}^{s}(1+b_i)^{-\alpha}$$

since $\sum_{i=2}^{s}(1+b_i)^{-\alpha}\leq K_\lambda$.

If we have a collision at step 1, then by the proof of Lemma 6.3 at step 2, $B^+\geq\sum_{i=1}^{s}c_\lambda b_i^2/(b_i+1)$. The value of b_i might have increased, but since $b_i^2/(b_i+1)$ is increasing in b_i, this would only make the inequality stronger. Thus the total expected increase in B over the two steps is at least

$$\left(c_\lambda\sum_{i=1}^{s}(1+b_i)^{-\alpha}\right)\left(c_\lambda\sum_{i=1}^{s}\frac{b_i^{2\alpha}}{(1+b_i)^{\alpha}}\right)\geq\frac{c_\lambda s^2}{4}.$$

Here we used the calculation from the proof of Lemma 6.3. By the same estimates for Q^- and B^- as in the proof of Lemma 6.3, the lemma follows. □

Finally, we will combine these results and with the aid of the results of §5 obtain the instability of inverse backoff.

LEMMA 6.5. *Let the system be at any state at time t. Then* $E\left(POT(t+\frac{N}{10})-POT(t)\right)\geq c_\lambda N^{5/2}$ *for sufficiently large N.*

Proof. Observe that by the previous lemmas, whenever $s>cN^{3/4}$, the expected increase per timestep in the potential is $\Omega(N^{3/2})$. To prove the lemma, we need only establish that with high probability, $s\geq cN^{3/4}$ during most of the interval. We have two cases. Remember that $s=N-x_e(t)$.

Case 1. $x_e(t)\leq\frac{4N}{5}$. Since at most one queue can become empty at each timeslot, the number of nonempty queues remains large during the entire interval.

Case 2. $x_e(t)\geq\frac{4N}{5}$. By Lemma 4.1, it follows that for any $r\geq N^{4/5}$, $\Pr[x_e(t+r)\geq N-\frac{\lambda r}{2}]\leq O(N^{-4/5})$, the reason being that to have many empty queues, either there has been many successful transmissions or not too many messages have arrived. The probability of the first event is small by Lemma 4.1 and the second probability is easily seen to be exponentially small. Using this, we get

$$E\left(POT\left(t+\frac{N}{10}\right)-POT(t)\right)\geq\sum_{r=1}^{\frac{N}{10}}\Delta POT(t+r)$$

$$\geq N^{\frac{4}{5}}\times(-N^{\frac{3}{2}})+\sum_{r=N^{\frac{4}{5}}}^{\frac{N}{10}}\Delta POT(t+r)\geq -N^{\frac{23}{10}}+c_\lambda N^{\frac{5}{2}}-c_\lambda N^{-\frac{4}{5}}N^{\frac{5}{2}}\geq c_\lambda N^{\frac{5}{2}},$$

and the lemma follows. □

Having established that the potential is expected to increase, we now prove Theorem 6.1. Observe first that to prove that the expected queue size becomes unbounded over time is trivial since the total queue size is always at least $POT/N^{3/2}$. We establish infinite expected recurrence time in the same way as in §5. To make the same argument go through, we only have to establish the lemma below.

TABLE 1
Observed values for L_{ave} after 10 million iterations of linear backoff.

N, λ	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8
2	0.02	0.12	0.44	1.4	5.5	24	97	420
5	0.03	0.24	1.1	9.0	$1.3 \cdot 10^5$	$6.4 \cdot 10^5$	$1.1 \cdot 10^6$	$1.6 \cdot 10^6$
10	0.04	0.29	1.9	$2.8 \cdot 10^5$	$7.8 \cdot 10^5$	$1.3 \cdot 10^6$	$1.8 \cdot 10^6$	$2.3 \cdot 10^6$
30	0.04	0.35	$4.4 \cdot 10^5$	$9.3 \cdot 10^5$	$1.4 \cdot 10^6$	$1.9 \cdot 10^6$	$2.4 \cdot 10^6$	$2.9 \cdot 10^6$
100	0.04	$3.7 \cdot 10^5$	$9.2 \cdot 10^5$	$1.4 \cdot 10^6$	$1.9 \cdot 10^6$	$2.4 \cdot 10^6$	$2.9 \cdot 10^6$	$3.4 \cdot 10^6$
300	0.04	$6.8 \cdot 10^5$	$1.2 \cdot 10^6$	$1.7 \cdot 10^6$	$2.2 \cdot 10^6$	$2.7 \cdot 10^6$	$3.2 \cdot 10^6$	$3.7 \cdot 10^6$

TABLE 2
Observed values for L_{ave} after 10 million iterations of quadratic backoff.

N, λ	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8
2	0.04	0.31	1.4	6.5	26	79	230	810
5	0.06	0.51	3.1	26	160	610	1800	5800
10	0.07	0.55	3.6	51	840	19000	$1.3 \cdot 10^5$	$3.8 \cdot 10^5$
30	0.07	0.55	3.6	470	$3.8 \cdot 10^5$	$8.7 \cdot 10^5$	$1.4 \cdot 10^6$	$1.9 \cdot 10^6$
100	0.07	0.52	3.5	$3.4 \cdot 10^5$	$8.4 \cdot 10^5$	$1.3 \cdot 10^6$	$1.8 \cdot 10^6$	$2.3 \cdot 10^6$
300	0.07	0.53	3.5	$7.0 \cdot 10^5$	$1.2 \cdot 10^6$	$1.7 \cdot 10^6$	$2.2 \cdot 10^6$	$2.7 \cdot 10^6$

LEMMA 6.6. *Assume that $N > N_\gamma$; then there are constants c and d such that for each T, the probability that the potential at time T is at least dT is bounded from below by c.*

Proof. The expected potential at time T is δT. On the other hand,

$$POT \leq N^{\frac{3}{2}} \sum q_i \leq N^{\frac{5}{2}} T.$$

Thus $\Pr[POT > \frac{\delta T}{2}] \geq \delta T / 2N^{5/2}$. □

Now the same argument as in §5 completes the proof of Theorem 6.1. □

Remark. Observe that in the proof of instability in the case of $f(i) = 2^{-i}$, we added a function depending on the backoff counters to the potential, while in the case of $f(i) = (i+1)^{-1}$, we subtracted a function. This reflects a basic fact, namely that in the exponential case, we back off too far, while in the inverse case, we back off too little.

7. Experimental results. We have simulated several backoff protocols for several different values of N and λ and with different initial conditions. The experiments were done rather for exploratory reasons rather than scientific investigation (with careful design and analysis of the experiments, use of strong pseudorandom number generators, etc.). In Tables 1–3, we present some of the results when the system starts with all empty queues. The values presented are for L_{ave} as computed over 10 million steps of the system. Each station was assumed to have arrival rate $\frac{\lambda}{N}$, where λ varied between 0.1 and 0.8 and N varied between 2 and 300. We emphasize that we have only done one experiment for each set of parameter values and we have not done any careful analysis of the data. Thus we leave it to the reader to interpret the data in any way. We also encourage the reader to design a careful experiment to evaluate the various protocols in practice. We think this would be of great interest.

8. Remarks. A natural question is for what backoff functions can we prove Theorem 3.1. We believe that it can be extended to any natural backoff function $f(x)$ for which

$$\int_{x=1}^{\infty} f(x)dx < \infty$$

TABLE 3
Observed values for L_{ave} after 10 million iterations of exponential backoff.

N, λ	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8
2	0.15	0.62	3.3	460	$2.3 \cdot 10^5$	$6.4 \cdot 10^5$	$1.1 \cdot 10^6$	$1.6 \cdot 10^6$
5	0.17	0.99	180	98000	$5.1 \cdot 10^5$	$7.1 \cdot 10^5$	$1.2 \cdot 10^6$	$1.7 \cdot 10^6$
10	0.17	0.99	96	$1.4 \cdot 10^5$	$3.5 \cdot 10^5$	$7.6 \cdot 10^5$	$1.2 \cdot 10^6$	$1.7 \cdot 10^6$
30	0.17	1.0	1200	$1.4 \cdot 10^5$	$4.9 \cdot 10^5$	$8.3 \cdot 10^5$	$1.3 \cdot 10^6$	$1.8 \cdot 10^6$
100	0.17	.95	610	$1.8 \cdot 10^5$	$5.1 \cdot 10^5$	$9.3 \cdot 10^5$	$1.4 \cdot 10^6$	$1.9 \cdot 10^6$
300	0.17	0.80	120	$1.9 \cdot 10^5$	$5.6 \cdot 10^5$	$1.0 \cdot 10^6$	$1.5 \cdot 10^6$	$1.9 \cdot 10^6$

and

$$\frac{d}{dx}[f(x)^{-1}] << f(x)^{-1}.$$

Note that integrating the second condition implies

$$f(x)^{-1} << \int_{b=1}^{x} f(b)^{-1}.$$

The first condition is necessary for proving that some station can dominate the channel for a long interval without ever losing control. Note that this condition is not satisfied for linear backoff but is satisfied for more rapid backoff protocols. The second condition is necessary to argue that the expected benefit from decreasing a large backoff counter outweighs the expected damage from backing off the counter further. It is also necessary to insure that the other counters aren't backing off too far when one station is dominating the channel. Note that this condition is not satisfied for exponential backoff but is satisfied for protocols that backoff less swiftly. A candidate for a potential function to use in such a stability result would be to pick a function $g(x)$ which grows faster than $f(x)^{-1}$ but slower than

$$\sum_{b=1}^{x} f(b)^{-1}$$

and then use a potential which is roughly

$$\sum_{i=1}^{n} q_i + \sum_{i=1}^{n} g(b_i).$$

Based on our analysis, it would appear that the most popular and well-studied protocols are precisely the wrong protocols. The good protocols, it would seem, are the protocols in between.

Although we have made substantial progress in analyzing the performance of backoff protocols for communication in multiple access channels, we also leave several open questions. Most importantly, it would be nice to determine the behavior of $\text{Ex}[L_{\text{ave}}]$ as a function of N and $1 - \lambda$ for polynomial backoff protocols. In particular, it would appear that the upper bounds are most in need of improvement. Once this is done, it might then be possible to decide which polynomial backoff protocol is best (i.e., which minimizes $\text{Ex}[L_{\text{ave}}]$ for a particular λ and N).

It would also be nice to completely determine the range of stability for exponential backoff.

Appendix. We do the calculation omitted in §5.

We need to prove that

$$C\lambda + M + 1 + \sum_{i=M+2}^{N} \lambda_i \geq (C+1)W + \delta,$$

where quantities are as defined in the beginning of §5. Now we must make use of the fact that $\lambda_i \geq \frac{\lambda}{N}$ for all i. Setting $C = 2N - 1$, it then suffices to prove that

$$(2N-1)\lambda + M + 1 + \frac{(N-M-1)\lambda}{N}$$
$$\geq 2N\left(1 - \frac{\lambda}{N}\right)^{N-M-1} + \delta.$$

For $\lambda > \lambda_0 + \frac{1}{4N-2}$, where $\lambda_0 = e^{-\lambda_0} \approx 0.567$, the preceding inequality holds for any sufficiently small constant $\delta > 0$. To prove this, it is sufficient to show that $g(M) > 0$ for $0 \leq M \leq N-1$ and $\lambda > \lambda_0 + \frac{1}{4N-2}$, where

$$\begin{aligned} g(M) &= (2N-1)\lambda + M + 1 + \frac{(N-M-1)\lambda}{N} - 2N\left(1 - \frac{\lambda}{N}\right)^{N-M-1} \\ &= 2N\lambda + (M+1)\left(1 - \frac{\lambda}{N}\right) - 2N\left(1 - \frac{\lambda}{N}\right)^{N-M-1}. \end{aligned}$$

We can then choose δ to be the minimum of $g(M)$ over $0 \leq M \leq N-1$.

We can show that $g(M)$ is always positive by proving that $g(0) > 0$ and $g(N-1) > 0$, and that $(d^2/dM^2)g(M) < 0$ for $0 \leq M \leq N-1$. The only difficult part is showing that $g(0) > 0$, so we save it for last.

We start by showing that $g(N-1) > 0$. This is easy since

$$g(N-1) = 2N\lambda + N - \lambda - 2N = \lambda(2N-1) - N$$

is positive provided that

$$\lambda > \frac{N}{2N-1} = \frac{1}{2} + \frac{1}{4N-2}.$$

We next show that $\frac{\delta^2}{\delta M^2} g(M) < 0$ for $0 \leq M \leq N-1$. This is also easy since

$$\frac{\delta}{\delta M} g(M) = \left(1 - \frac{\lambda}{N}\right) + 2N \ln\left(1 - \frac{\lambda}{N}\right)\left(1 - \frac{\lambda}{N}\right)^{N-M-1}$$

and

$$\frac{\delta^2}{\delta M^2} g(M) = -2N\left(\ln\left(1 - \frac{\lambda}{N}\right)\right)^2 \left(1 - \frac{\lambda}{N}\right)^{N-M-1}.$$

Lastly, we must prove that $g(0) > 0$. The argument here is a bit trickier because we must be careful when bounding the $(1 - \frac{\lambda}{N})^{N-1}$ term in the expression for

$$g(0) = 2N\lambda + 1 - \frac{\lambda}{N} - 2N\left(1 - \frac{\lambda}{N}\right)^{N-1}.$$

We start by observing that

$$\left(1-\frac{\lambda}{N}\right)=e^{-\frac{\lambda}{N}-\frac{\lambda^2}{2N^2}-\frac{\lambda^3}{3N^2}-\cdots}$$

and thus that

$$\begin{aligned}\left(1-\frac{\lambda}{N}\right)^{N-1} &= e^{\lambda+\frac{\lambda}{N}-\frac{\lambda^2}{2N}+\frac{\lambda^2}{2N^2}-\frac{\lambda^3}{3N^2}+\frac{\lambda^3}{3N^3}-\cdots}\\ &= e^{-\lambda+\frac{\lambda(2-\lambda)}{2N}+\frac{\lambda^2(3-2\lambda)}{6N^2}+\frac{\lambda^3(4-3\lambda)}{12N^3}+\cdots}\\ &\le e^{-\lambda+\frac{1}{2N}+\frac{1}{6N^2}+\frac{1}{12N^3}+\cdots}\\ &< e^{-\lambda+\frac{1}{2N}+\frac{1}{6N^2}\left(\frac{1}{1-1/N}\right)}\\ &\le e^{-\lambda+\frac{1}{2N-1}}\\ &= e^{-(\lambda-\frac{1}{4N-2})+\frac{1}{4N-2}}\\ &\le \left(\lambda-\frac{1}{4N-2}\right)e^{\frac{1}{4N-2}}\\ &\le \left(\lambda-\frac{1}{4N-2}\right)\frac{1}{1-\frac{1}{4N-2}}\\ &\le \lambda\end{aligned}$$

since $\lambda - \frac{1}{4N-2} > \lambda_0$ and $e^{-\lambda'} \le \lambda'$ for all $\lambda' \ge \lambda_0$, and $e^x \le \frac{1}{1-x}$ for $0 \le x < 1$. Hence

$$g(0) \ge 2N\lambda + 1 - \frac{\lambda}{N} - 2N\lambda > 0.$$

Acknowledgments. We are deeply indebted to Albert Greenberg for putting us straight with respect to the various definitions of stability. His comments saved us from potentially serious errors in the argument. We also want to thank Richard Koch for many discussions and suggestions and Richard Ladner, Robert Maier, Ron Rivest, Wojciech Szpankowski, and John Tsitsiklis for their helpful comments and references. We are also grateful to one of the referees for pointing us to the book [8].

REFERENCES

[1] D. J. Aldous, *Ultimate instability of exponential backoff protocol for acknowledgement based transmission control of random access communication channels*, IEEE Trans. Inform. Theory, IT-33 (1987), pp. 219–223.

[2] J. Goodman, A. Greenberg, N. Madras, and P. March, *On the stability of the Ethernet*, in Proc. 17th ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1985, pp. 379–387.

[3] B. Hajek and T. van Loon, *Decentralized dynamic control of a multiaccess broadcast channel*, IEEE Trans. Automat. Control, AC-27 (1982), pp. 559–569.

[4] *IEEE Trans. Inform. Theory*, IT-31 (1985).

[5] F. P. Kelley, *Stochastic models of computer communications systems*, J. Roy. Statist. Soc. Ser. B, 47 (1985), pp. 379–395.

[6] R. Ladner, personal communication, 1986.

[7] R. Metcalf and D. Boggs, *Ethernet: Distributed packet switching for local computer networks*, Comm. Assoc. Comput. Mach., 19 (1976), pp. 395–404.

[8] S. P. Meyn and R. L. Tweedie, *Markov Chains and Stochastic Stability*, Springer-Verlag, London, 1993.

[9] R. Rivest, *Network control by Bayesian broadcast*, IEEE Trans. Inform. Theory, IT-33 (1987), pp. 323–328.

[10] W. Rosenkrantz, *Some theorems on the instability of the exponential backoff protocol*, in Proc. 10th International Symposium on Computer Performance, E. Gelenke, ed., Elsevier, Amsterdam, 1984.

[11] J. Shoch and J. Hupp, *Measured performance of an Ethernet local network*, Comm. Assoc. Comput. Mach., 23 (1980), pp. 711–721.

[12] S. Stidham, *The last word on the* $L = \lambda W$, Oper. Res., 22 (1974), pp. 417–421.

[13] W. Szpankowski, *Stability conditions for multidimensional queuing systems with applications*, Oper. Res., 36 (1988), pp. 944–957.

[14] W. Szpankowski and V. Rego, *Some theorems on instability with applications to multiaccess protocols*, Oper. Res., 36 (1988), pp. 958–966.

[15] A. Tanenbaum, *Network protocols*, Comput. Surveys, 13 (1981), pp. 453–489.

[16] J. Tsitsiklis, *Analysis of a multiaccess control scheme*, IEEE Trans. Automat. Control, AC-32 (1987), pp. 1017–1020.

[17] B. Tsybakov and V. Mikhailov, *Ergodicity of a slotted Aloha system*, Problems Inform. Transmission, 15 (1980) (translated version), pp. 301–312.

SIAM J. COMPUT.
Vol. 25, No. 4, pp. 775–796, August 1996

NEW TECHNIQUES FOR EXACT AND APPROXIMATE DYNAMIC CLOSEST-POINT PROBLEMS*

SANJIV KAPOOR[†] AND MICHIEL SMID[‡]

Abstract. Let S be a set of n points in $\mathbb{R}^D$. It is shown that a range tree can be used to find an L_∞-nearest neighbor in S of any query point in $O((\log n)^{D-1} \log\log n)$ time. This data structure has size $O(n(\log n)^{D-1})$ and an amortized update time of $O((\log n)^{D-1} \log\log n)$. This result is used to solve the $(1+\epsilon)$-approximate L_2-nearest-neighbor problem within the same bounds (up to a constant factor that depends on ϵ and D). In this problem, for any query point p, a point $q \in S$ is computed such that the euclidean distance between p and q is at most $(1+\epsilon)$ times the euclidean distance between p and its true nearest neighbor. This is the first dynamic data structure for this problem having close to linear size and polylogarithmic query and update times.

New dynamic data structures are given that maintain a closest pair of S. For $D \geq 3$, a structure of size $O(n)$ is presented with amortized update time $O((\log n)^{D-1} \log\log n)$. The constant factor in this space (resp. time bound) is of the form $O(D)^D$ (resp. $2^{O(D^2)}$). For $D = 2$ and any nonnegative integer constant k, structures of size $O(n \log n/(\log\log n)^k)$ (resp. $O(n)$) are presented that have an amortized update time of $O(\log n \log\log n)$ (resp. $O((\log n)^2/(\log\log n)^k)$). Previously, no deterministic linear size data structure having polylogarithmic update time was known for this problem.

Key words. proximity, dynamic data structures, point location, approximation

AMS subject classification. 68U05

1. Introduction.

Closest-point problems are among the basic problems in computational geometry. In such problems, a set S of n points in D-dimensional space is given and we have to store it in a data structure such that a point in S nearest to a query point can be computed efficiently, or we have to compute a closest pair in S or, for each point in S, another point in S that is closest to it. These problems are known as the *nearest-neighbor problem*, the *closest-pair problem*, and the *all-nearest-neighbors problem*, respectively. In the dynamic version of these problems, the set S is changed by insertions and deletions of points.

It is assumed that the dimension $D \geq 2$ is a constant independent of n. Moreover, distances are measured in the L_t-metric, where $1 \leq t \leq \infty$ is a fixed real number. In this metric, the distance $d_t(p, q)$ between the points p and q is defined as $d_t(p,q) = (\sum_{i=1}^{D} |p_i - q_i|^t)^{1/t}$ if $1 \leq t < \infty$, and for $t = \infty$ it is defined as $d_\infty(p,q) = \max_{1 \leq i \leq D} |p_i - q_i|$.

The planar version of the nearest-neighbor problem can be solved optimally, i.e., with $O(\log n)$ query time using $O(n)$ space, by means of Voronoi diagrams. (See [15].) In higher dimensions, however, the situation is much worse. The best results known are due to Clarkson [7] and Arya et al. [1]. In [7], a randomized data structure is given that finds a nearest neighbor of a query point in $O(\log n)$ expected time. This structure has size $O(n^{\lceil D/2 \rceil + \delta})$, where δ is an arbitrarily small positive constant. In [1], the problem is solved with an expected query time of $O(n^{1-1/\lceil (D+1)/2 \rceil} (\log n)^{O(1)})$ using $O(n \log\log n)$ space.

It seems that in higher dimensions, it is impossible to obtain polylogarithmic query time using $O(n(\log n)^{O(1)})$ space. Moreover, even in the planar case, there is no dynamic data structure known that has polylogarithmic query and update times and that uses $O(n(\log n)^{O(1)})$ space.

*Received by the editors November 19, 1993; accepted for publication (in revised form) December 3, 1994.

[†]Department of Computer Science, Indian Institute of Technology, Hauz Khas, New Delhi 110016, India (skapoor@cse.iitd.ernet.in). The research on this work began when this author visited the Max-Planck-Institut für Informatik.

[‡]Max-Planck-Institut für Informatik, Im Stadtwald, D-66123 Saarbrücken, Germany (michiel@mpi-sb.mpg.de). The research of this author was supported by ESPRIT Basic Research Actions Program contract 7141 (project ALCOM II).

TABLE 1

Deterministic data structures for the dynamic closest-pair problem. In the last two lines, k is an arbitrary nonnegative integer constant. All bounds are "big-oh" with constant factors that depend on the dimension D and, in the last two lines, on D and k. The update times are either worst-case (w) or amortized (a).

mode	dimension	update time	space	reference
insertions	$D \geq 2$	$\log n$ (w)	n	[17, 18]
deletions	$D \geq 2$	$(\log n)^D$ (a)	$n(\log n)^{D-1}$	[23]
dynamic	$D \geq 2$	$\sqrt{n} \log n$ (w)	n	[16, 20]
dynamic	$D \geq 2$	$(\log n)^D \log\log n$ (a)	$n(\log n)^D$	[22]
dynamic	$D \geq 3$	$(\log n)^{D-1} \log\log n$ (a)	n	this paper
dynamic	2	$\log n \log\log n$ (a)	$n \log n/(\log\log n)^k$	this paper
dynamic	2	$(\log n)^2/(\log\log n)^k$ (a)	n	this paper

Therefore, it is natural to ask whether the *approximate nearest-neighbor problem* allows more efficient solutions. Let $\epsilon > 0$. A point $q \in S$ is called a $(1+\epsilon)$-approximate neighbor of a point $p \in \mathbb{R}^D$ if $d_t(p,q) \leq (1+\epsilon)d_t(p,p^*)$, where $p^* \in S$ is the true nearest-neighbor of p.

This approximate neighbor problem was considered in Bern [5] and Arya et al. [1, 2]. In the latter paper, the problem is solved optimally: They give a deterministic data structure of size $O(n)$ that can find a $(1+\epsilon)$-approximate neighbor in $O(\log n)$ time, for any positive constant ϵ. At this moment, however, no dynamic data structures are known for this problem.

In this paper, we first show that for the L_∞-metric, the nearest-neighbor problem can be solved efficiently. To be more precise, we show that the range tree (see [12, 14, 15, 25]) can be used to solve this problem. As a result, we solve the L_∞-neighbor problem with a query time of $O((\log n)^{D-1} \log\log n)$ and an amortized update time of $O((\log n)^{D-1} \log\log n)$ using $O(n(\log n)^{D-1})$ space. For the static version of this problem, the query time is $O((\log n)^{D-1})$ and the space bound is $O(n(\log n)^{D-2})$.

Using this result, we give a data structure that solves the approximate L_2-neighbor problem, for any positive constant ϵ, within the same complexity bounds. (The constant factors depend on D and ϵ.)

We also consider the dynamic closest-pair problem. Note that the static version has been solved already for a long time. Several algorithms are known that compute a closest pair in $O(n \log n)$ time, which is optimal. (See [4, 10, 15, 19].) The dynamic version, however, has been investigated only recently. In Table 1, we give an overview of the currently best-known data structures for maintaining a closest pair under insertions and/or deletions of points. For an up-to-date survey, we refer the reader to Schwarz's Ph.D. Thesis [17].

Note that all data structures of Table 1 are deterministic and can be implemented in the algebraic computation tree model. If we add randomization to this model, then there is a data structure of size $O(n)$ that maintains a closest pair in $O((\log n)^2)$ expected time per update. (See Golin et al. [9].)

In this paper, we give new deterministic data structures for the dynamic closest-pair problem. These structures are based on our solution to the L_∞-neighbor problem, data structures for maintaining boxes of constant overlap, and a new transformation. Given *any* dynamic closest-pair data structure having more than linear size, this transformation produces another dynamic closest-pair structure that uses less space. The complexity bounds of the new structures are shown in the last three lines of Table 1. The results of the last two lines are obtained by applying the transformation repeatedly. Note that we obtain the first linear-size deterministic data structure that maintains a closest pair in polylogarithmic time.

Finally, we consider the all-nearest-neighbors problem. Again, in the planar case, the problem can be solved using Voronoi diagrams. For the D-dimensional case, Vaidya [24] has given an optimal $O(n \log n)$-time algorithm that computes the L_t-neighbor for each point of S. No nontrivial solutions seem to be known for maintaining each point's nearest neighbor. One of our data structures (not the ones mentioned in Table 1) for maintaining the closest pair basically maintains the L_∞-neighbor of each point in S. As a result, we get a data structure of size $O(n(\log n)^{D-1})$ that maintains for each point in S its L_∞-neighbor in $O((\log n)^{D-1} \log \log n)$ amortized time per update.

We want to remark here that all our algorithms use classical and well-understood data structures, such as range trees, segment trees, and skewer trees. Moreover, we apply the well-known technique of fractional cascading [6, 14] several times.

The rest of this paper is organized as follows. In §2, we recall the definition of a range tree. This data structure is used in §3 to solve the L_∞-neighbor problem. In §4, we give the data structure for solving the approximate L_2-nearest neighbor problem.

Our first data structure for maintaining the closest pair stores a dynamically changing set of boxes that are of constant overlap, i.e., there is a constant c such that each box contains the centers of at most c boxes in its interior. We have to maintain such boxes under insertions and deletions such that for any query point $p \in \mathbb{R}^D$, we can find all boxes that contain p. In §5, we give two data structures for this problem. The first one is based on segment trees. (See [13, 15].) Therefore, its size is superlinear. We also give a linear-size solution that has a slightly worse update time in the planar case. This solution is based on skewer trees. (See [8, 21].)

In §6, we use the results obtained to give a new data structure that maintains the closest pair in a point set. This structure has size $O(n(\log n)^{D-1})$ and an amortized update time of $O((\log n)^{D-1} \log \log n)$. It immediately gives the dynamic data structure for the all-L_∞-nearest-neighbors problem. In §7, we give the transformation that reduces the size of a dynamic closest-pair structure. Applying this transformation repeatedly to the structure of §6 gives the new results mentioned in Table 1. Some concluding remarks are given in §8.

2. Range trees. In this section, we recall the definition of a range tree. See [12, 13, 15, 25]. The coordinates of a point p in $\mathbb{R}^D$ are denoted by p_i, $1 \leq i \leq D$. Moreover, we denote by p' the point $(p_2, \ldots, p_D)$ in $\mathbb{R}^{D-1}$. If S is a set of points in $\mathbb{R}^D$, then we define $S' := \{p' : p \in S\}$. We note that S' is to be considered as a multiset, i.e., elements may occur more than once.

The range tree. Let S be a set of n points in $\mathbb{R}^D$. A *D-dimensional range tree* for the set S is defined as follows. For $D = 1$, it is a balanced binary search tree, storing the elements of S in sorted order in its leaves.

For $D > 1$, it consists of a balanced binary search tree, called the *main tree*, storing the points of S in its leaves, sorted by their first coordinates. For each internal node v of this main tree, let S_v be the set of points that are stored in its subtree. Node v contains a pointer to the rightmost leaf in its subtree and (a pointer to) an *associated structure*, which is a $(D-1)$-dimensional range tree for the set S'_v.

Hence, a two-dimensional range tree for a set S consists of a binary tree, storing the points of S in its leaves sorted by their x-coordinates. Each internal node v of this tree contains a pointer to the rightmost leaf in its subtree and (a pointer to) a binary tree that stores the points of S_v in its leaves sorted by their y-coordinates.

Let p be any point in $\mathbb{R}^D$. Consider the set $\{r \in S : r_1 \geq p_1\}$, i.e., the set of all points in S having a first coordinate that is at least equal to p's first coordinate. Using the range tree, we can decompose this set into $O(\log n)$ canonical subsets:

Initialize $M := \emptyset$. Starting in the root of the main tree, search for the leftmost leaf storing a point whose first coordinate is at least equal to p_1. During this search, each time we move from a node v to its left son, add the right son of v to the set M. Let v be the leaf in which this search ends. If the point stored in this leaf has a first coordinate that is at least equal to p_1, then add v to the set M.

LEMMA 1. *The set M of nodes of the main tree that is computed by the given algorithm satisfies*

$$\{r \in S : r_1 \geq p_1\} = \bigcup_{v \in M} S_v.$$

Moreover, M is computed in $O(\log n)$ time.

Range trees can be maintained under insertions and deletions of points such that each binary tree that is part of the structure has a height logarithmic in the number of its leaves. The update algorithms use dynamic fractional cascading. For details, we refer the reader to Mehlhorn and Näher [14]. In the following theorem, we state the complexity of the range tree.

THEOREM 1. *A D-dimensional range tree, where $D \geq 2$, for a set of n points has $O(n(\log n)^{D-1})$ size and can be maintained in $O((\log n)^{D-1} \log\log n)$ amortized time per insertion and deletion.*

3. The L_∞-neighbor problem. Recall the following notations. For any point $p = (p_1, p_2, \ldots, p_D) \in \mathbb{R}^D$, we denote by p' the point $(p_2, \ldots, p_D)$ in $\mathbb{R}^{D-1}$. If S is a set of points in $\mathbb{R}^D$, then $S' = \{p' : p \in S\}$. Finally, if v is a node of the main tree of a range tree storing a set S, then S_v denotes the set of points that are stored in the subtree of v.

Let S be a set of n points in $\mathbb{R}^D$. We want to store this set into a data structure such that for any query point $p \in \mathbb{R}^D$, we can find a point in S having minimal L_∞-distance to p. Such a point is called an *L_∞-neighbor* of p in S.

Let q be an L_∞-neighbor of p in the set $\{s \in S : s_1 \geq p_1\}$. We call q a *right-L_∞-neighbor* of p in S. Similarly, a point r is called a *left-L_∞-neighbor* of p in S if it is an L_∞-neighbor of p in the set $\{s \in S : s_1 \leq p_1\}$. In order to guarantee that both neighbors always exist, we add the $2D$ artificial points $(a_1, \ldots, a_D)$, where all a_i are zero except for one which is either ∞ or $-\infty$, to the set S. Note that none of these points can be L_∞-neighbor of any query point.

The data structure that solves the L_∞-neighbor problem is just a D-dimensional range tree storing the points of S and the artificial points. In Figure 1, our recursive algorithm is given that finds an L_∞-neighbor of any query point in $\mathbb{R}^D$.

We prove the correctness of this query algorithm. It is clear that the algorithm is correct in the one-dimensional case. So let $D \geq 2$ and assume that the algorithm is correct for smaller values of D. The following lemma turns out to be useful.

LEMMA 2. *Let $p \in \mathbb{R}^D$ and let q be its right-L_∞-neighbor in S. Let w be any node in the main tree of the range tree such that $q \in S_w$ and p has a first coordinate that is at most equal to the first coordinate of any point in S_w. Let r be the point of S that is stored in the rightmost leaf of w's subtree. Finally, let*

$$N := |\{x \in S_w : |p_j - x_j| \leq |p_1 - r_1|, 2 \leq j \leq D\}|.$$

If $N = 0$, then q' is an L_∞-neighbor of p' in the $(D-1)$-dimensional set S'_w.

Proof. The proof is by contradiction. So let s be a point in S_w such that s' is an L_∞-neighbor of p' in the set S'_w and assume that $d_\infty(p', s') < d_\infty(p', q')$. Let $\delta := |p_1 - r_1|$.

Since q and s are elements of S_w, we have $|p_1 - q_1| \leq \delta$ and $|p_1 - s_1| \leq \delta$. Since $N = 0$, there is a j, $2 \leq j \leq D$, such that $|p_j - q_j| > \delta$. Therefore, $d_\infty(p, q) = d_\infty(p', q')$. Similarly, there is a k, $2 \leq k \leq D$, such that $|p_k - s_k| > \delta$ and hence $d_\infty(p, s) = d_\infty(p', s')$. This implies that $d_\infty(p, s) < d_\infty(p, q)$, i.e., q is not a right-L_∞-neighbor of p in S. This is a contradiction. □

```
ALGORITHM NEIGHBOR($p$, $S$, $D$)   (* returns an $L_\infty$-neighbor of $p$ in $S$ *)
begin
1. Assume $D = 1$.
   $q^L$ := maximal element in the one-dimensional range tree that is less than $p$;
   $q^R$ := minimal element in the one-dimensional range tree that is at least
          equal to $p$;
   if $|p - q^R| \le |p - q^L|$ then return $q^R$ else return $q^L$ fi;
2. Assume $D \ge 2$.
2a. Compute a right-$L_\infty$-neighbor:
    Stage 1. Compute the set $M$ of nodes of the main tree, see Lemma 1.
          Number these nodes $v_1, v_2, \ldots, v_m$, $m = |M|$, where $v_i$ is closer to
          the root than $v_{i-1}$, $2 \le i \le m$.
    Stage 2. (* one of the sets $S_{v_i}$ contains a right-$L_\infty$-neighbor of $p$ *)
          $C := \emptyset$; $i := 1$; $stop := false$;
          while $i \le m$ and $stop = false$
          do $x' :=$ Neighbor($p'$, $S'_{v_i}$, $D - 1$);
             $r$ := the point stored in the rightmost leaf of the subtree of $v_i$;
             if $d_\infty(p', x') > |p_1 - r_1|$
             then $C := C \cup \{x\}$; $i := i + 1$
             else $v := v_i$; $stop := true$
             fi
          od;
          if $stop = false$
          then $q^R$ := a point of $C$ having minimal $L_\infty$-distance to $p$;
               goto 2b
          fi;
    Stage 3. (* the set $C \cup S_v$ contains a right-$L_\infty$-neighbor of $p$ *)
          while $v$ is not a leaf
          do $w$ := left son of $v$;
             $x' :=$ Neighbor($p'$, $S'_w$, $D - 1$);
             $r$ := the point stored in the rightmost leaf of the subtree of $w$;
             if $d_\infty(p', x') > |p_1 - r_1|$
             then $C := C \cup \{x\}$; $v$ := right son of $v$
             else $v := w$
             fi
          od;
          $q^R$ := a point of $C \cup S_v$ having minimal $L_\infty$-distance to $p$;
2b. Compute a left-$L_\infty$-neighbor:
    In a completely symmetric way, compute a left-$L_\infty$-neighbor $q^L$ of $p$;
2c. if $d_\infty(p, q^R) \le d_\infty(p, q^L)$ then return $q^R$ else return $q^L$ fi
end
```

FIG. 1. *Finding an L_∞-neighbor.*

We analyze part **2a** of the algorithm. As we will see, this part computes a right-L_∞-neighbor of p. Consider the nodes $v_1, \ldots, v_m$ that are computed in Stage 1. We know that

$$\{r \in S : r_1 \ge p_1\} = \bigcup_{i=1}^{m} S_{v_i}.$$

Hence one of the sets S_{v_i} contains a right-L_∞-neighbor of p.

Note that each point in $S_{v_{i-1}}$ has a first coordinate that is at most equal to the first coordinate of any point in S_{v_i}, $2 \leq i \leq m$.

LEMMA 3. *If the variable stop has value false after the while-loop of Stage* 2 *has been completed, then the set C contains a right-L_∞-neighbor of p in S.*

Proof. Let i, $1 \leq i \leq m$, be an index such that the set S_{v_i} contains a right-L_∞-neighbor q of p. We consider what happens during the ith iteration of the while-loop. Let x and r be the points that are selected in this iteration. Since $d_\infty(p', x') > |p_1 - r_1|$, it is clear that the value of the integer

$$N := |\{y \in S_{v_i} : |p_j - y_j| \leq |p_1 - r_1|, 2 \leq j \leq D\}|$$

is zero. Then Lemma 2 implies that q' is an L_∞-neighbor of p' in S'_{v_i}. Hence $d_\infty(p', x') = d_\infty(p', q')$. Since $d_\infty(p', x') = d_\infty(p, x)$ and $d_\infty(p', q') = d_\infty(p, q)$, it follows that $d_\infty(p, x) = d_\infty(p, q)$. This proves that x is a right-L_∞-neighbor of p. Since x is added to C during this iteration, the proof is completed. □

LEMMA 4. *If the variable stop has value true after the while-loop of Stage* 2 *has been completed, then the set $C \cup S_v$ contains a right-L_∞-neighbor of p in S.*

Proof. Consider the integer i such that during the ith iteration of the while-loop, the variable *stop* is set to the value *true*. Then $v = v_i$.

We distinguish two cases. First let $j \geq i + 1$ and assume that the set S_{v_j} contains a right-L_∞-neighbor q of p in S. Consider what happens during the ith iteration of the while-loop. Let x and r be the points that are selected during this iteration. Moreover, let $\delta := |p_1 - r_1|$. Since $|p_1 - x_1| \leq \delta$ and $d_\infty(p', x') \leq \delta$, we have $d_\infty(p, x) \leq \delta$. Since $q \in S_{v_j}$, we have

$$d_\infty(p, q) \geq q_1 - p_1 \geq r_1 - p_1 = \delta.$$

This implies that $d_\infty(p, x) \leq d_\infty(p, q)$. Hence, x is also a right-L_∞-neighbor of p. Since $x \in S_v$, the claim of the lemma follows.

Next, let $1 \leq j \leq i$ and assume that the set S_{v_j} contains a right-L_∞-neighbor of p in S. If $j = i$, then we are done. If $j < i$, then in the same way as in the proof of Lemma 3, it follows that such a neighbor is added to C during the jth iteration of the while loop. This completes the proof. □

LEMMA 5. *During the while-loop of Stage* 3, *the set $C \cup S_v$ contains a right-L_∞-neighbor of p in S.*

Proof. The previous lemma implies that the claim holds before the while-loop is entered. Consider an iteration and assume that $C \cup S_v$ contains a right-L_∞-neighbor of p. (We consider C as it is at the beginning of this iteration.) Let x and r be the points that are selected during this iteration. Moreover, let w_l (resp. w_r) be the left (resp. right) son of v.

First assume that $d_\infty(p', x') > |p_1 - r_1|$. We have to show that $C \cup \{x\} \cup S_{w_r}$ contains a right-L_∞-neighbor of p. Since the set $C \cup S_{w_l} \cup S_{w_r}$ contains a right-L_∞-neighbor of p, we only have to consider the case where the set S_{w_l} contains such a neighbor. In this case, it follows in the same way as in the proof of Lemma 3 that point x is a right-L_∞-neighbor of p. Since this point is added to C during this iteration, the claim follows.

It remains to consider the case where $d_\infty(p', x') \leq |p_1 - r_1|$. Now we have to show that the set $C \cup S_{w_l}$ contains a right-L_∞-neighbor of p. Assume that S_{w_r} contains such a neighbor. As in the proof of the previous lemma, it follows that x is also a right-L_∞-neighbor of p. However, x is an element of S_{w_l}. This completes the proof. □

Lemmas 3 and 5 imply that point q^R which is computed in part **2a** is a right-L_∞-neighbor of p in S. Similarly, point q^L computed in part **2b** is a left-L_∞-neighbor of p. This proves that the point that is returned in part **2c** is an L_∞-neighbor of p. Hence algorithm Neighbor(p, S, D) is correct.

LEMMA 6. *For $D \geq 2$, the running time of algorithm* Neighbor(p, S, D) *is bounded by* $O((\log n)^{D-1})$.

Proof. Let $Q(n, D)$ denote the running time on a set of n points in $\mathbb{R}^D$. It is clear that $Q(n, 1) = O(\log n)$. Let $D \geq 2$. Consider part **2a**. Stage 1 takes $O(\log n)$ time. The while-loop in Stage 2 takes time $O(Q(n, D-1) \log n)$. If *stop* = *false* after this loop, then $O(|C|) = O(\log n)$ time is needed to select the point q^R. Hence Stage 2 takes time $O(Q(n, D-1) \log n)$. Stage 3 takes the same amount of time. Part **2b** can be analyzed in the same way. Clearly, part **2c** takes only constant time. This proves that $Q(n, D) = O(Q(n, D-1) \log n)$, implying that $Q(n, D) = O((\log n)^D)$.

Now consider the planar case. The algorithm follows a path in the main tree and locates the y-coordinate of the query point in the associated structure—a binary search tree—of some of the nodes on this path. It is well known that layering or fractional cascading (see [6, 15]) can be applied to improve the query time from $O((\log n)^2)$ to $O(\log n)$, i.e., $Q(n, 2) = O(\log n)$. As a result, the query time for the D-dimensional case, where $D \geq 2$, is improved to $O((\log n)^{D-1})$. □

Hence, we have a data structure for the L_∞-neighbor problem that has a query time of $O((\log n)^{D-1})$ and that uses $O(n(\log n)^{D-1})$ space. We can improve the space bound by noting that the planar version can be solved by means of the L_∞-Voronoi diagram. (See Lee [11].) That is, the planar L_∞-neighbor problem has a solution with a query time of $O(\log n)$ using only $O(n)$ space. As a result, the D-dimensional problem can be solved with a query time of $O((\log n)^{D-1})$ using $O(n(\log n)^{D-2})$ space.

Until now, we have only considered the static version of the problem. We saw in Theorem 1 that range trees can be maintained under insertions and deletions of points. Therefore, our solution that is based on the range tree can be used only for the dynamic problem. Because of dynamic fractional cascading, the query time increases by a factor of $O(\log \log n)$. This proves the following result.

THEOREM 2. *Let S be a set of n points in $\mathbb{R}^D$. There exists a data structure of size $O(n(\log n)^{D-2})$ that, given a query point $p \in \mathbb{R}^D$, finds an L_∞-neighbor of p in S in $O((\log n)^{D-1})$ time.*

For the dynamic version of the problem, there is a structure of size $O(n(\log n)^{D-1})$ having $O((\log n)^{D-1} \log \log n)$ query time and $O((\log n)^{D-1} \log \log n)$ amortized update time.

Remark. Let c be a positive integer. The range tree can also be used to compute the c L_∞-neighbors of a query point. The algorithm is basically the same. If c is a constant, the query time remains the same as in the above theorem.

4. The approximate L_2-neighbor problem. We mentioned in the introduction that the L_2-nearest-neighbor problem is hard to solve exactly. Therefore, it is natural to consider a weaker version of the problem.

Let S be a set of n points in $\mathbb{R}^D$ and let $\epsilon > 0$ be a fixed constant. For any point $p \in \mathbb{R}^D$, we denote by p^* its L_2-neighbor in S, i.e., p^* is a point of S such that $d_2(p, p^*) = \min\{d_2(p, q) : q \in S\}$. A point $q \in S$ is called a *$(1+\epsilon)$-approximate L_2-neighbor* of p if $d_2(p, q) \leq (1+\epsilon) d_2(p, p^*)$.

We want to store the set S in a data structure such that for any query point $p \in \mathbb{R}^D$, we can find a *$(1+\epsilon)$-approximate L_2-neighbor* of it.

Let $p \in \mathbb{R}^D$ and let q be an L_∞-neighbor of p in S. Let $\delta := d_\infty(p, p^*)$ and consider the D-dimensional axes-parallel box centered at p having sides of length 2δ. Clearly, q lies inside or on the boundary of this box. Therefore, $d_2(p, q) \leq \sqrt{D} \cdot \delta$. Since $\delta = d_\infty(p, p^*) \leq d_2(p, p^*)$, we infer that $d_2(p, q) \leq \sqrt{D} \cdot d_2(p, p^*)$, i.e., q is a $\sqrt{D}$-approximate L_2-neighbor of p.

This shows that we can solve the $\sqrt{D}$-approximate L_2-neighbor problem using the results of the previous section.

We extend this solution. First note that the L_∞-metric depends on the coordinate system: If we rotate the (XY)-system, then the L_∞-metric changes. The L_2-metric, however, is invariant under such rotations. We store the set S in a constant number of range trees, where each range tree stores the points according to its own coordinate system. Then, given p, we use the range trees to compute L_∞-neighbors in all coordinate systems. As we will see, one of these L_∞-neighbors is a $(1+\epsilon)$-approximate L_2-neighbor of p.

Let $(\mathcal{F}_i)$ be a family of orthonormal coordinate systems all sharing the same origin. The coordinates of any point x in the system $\mathcal{F}_i$ are denoted by $x_{i1}, x_{i2}, \ldots, x_{iD}$. Moreover, $d_{i,\infty}(\cdot,\cdot)$ denotes the L_∞-distance function in $\mathcal{F}_i$. Assume that for any point x in D-dimensional space, there is an index i such that for all $1 \leq j \leq D$,

$$(1) \qquad 0 \leq x_{ij} \leq x_{iD} \leq (1+\epsilon)x_{ij},$$

i.e., in $\mathcal{F}_i$ all coordinates of x are nonnegative and almost equal. Note that the family $(\mathcal{F}_i)$ is independent of the set S.

In Yao [26], it is shown how such a family consisting of $O((c/\epsilon)^{D-1})$ coordinate systems can be constructed. (Here c is a constant.) In the planar case, such a family is easily obtained: Let $0 < \phi < \pi/4$ be such that $\tan\phi = \epsilon/(2+\epsilon)$. For $0 \leq i < 2\pi/\phi$, let X_i (resp. Y_i) be the directed line that makes an angle of $i \cdot \phi$ with the positive X-axis (resp. Y-axis). Then $\mathcal{F}_i$ is the $(X_i Y_i)$-coordinate system.

To prove that (1) holds for this family, let x be any point in the plane and let γ be the angle that the line segment $\vec{x}$ between the origin and x makes with the positive X-axis.

First, assume that $\pi/4 \leq \gamma < 2\pi$. Let $i := \lfloor(\gamma - \pi/4)/\phi\rfloor$ and let γ_i be the angle between $\vec{x}$ and the positive X_i-axis. Then $\gamma_i = \gamma - i \cdot \phi$ and this angle lies in between $\pi/4$ and $\pi/4 + \phi$. Therefore, the coordinates x_{i1} and x_{i2} of x in $\mathcal{F}_i$ satisfy

$$x_{i2}/x_{i1} = \tan\gamma_i \geq \tan\pi/4 = 1$$

and

$$x_{i2}/x_{i1} = \tan\gamma_i \leq \tan(\pi/4+\phi) = \frac{1+\tan\phi}{1-\tan\phi} = 1+\epsilon.$$

If $0 \leq \gamma < \pi/4$, then we take $i := \lfloor(7\pi/4+\gamma)/\phi\rfloor$. In this case, $\gamma_i = 2\pi - i\cdot\phi + \gamma$ and this angle lies in between $\pi/4$ and $\pi/4+\phi$. This proves that x_{i2}/x_1 lies in between 1 and $1+\epsilon$.

Hence, the family $(\mathcal{F}_i)$ satisfies the assumptions made in (1). Moreover, the number of coordinate systems is at most

$$1 + \frac{2\pi}{\phi} = 1 + \frac{2\pi}{\arctan \epsilon/(2+\epsilon)} = O(1/\epsilon).$$

The data structure for approximate L_2-neighbor queries. For each index i, let S_i denote the set of points in S with coordinates in the system $\mathcal{F}_i$. The data structure consists of a collection of range trees; the ith range tree stores the set S_i.

Finding an approximate L_2-neighbor. Let $p \in \mathbb{R}^D$ be a query point. For each index i, use the ith range tree to find an L_∞-neighbor $q^{(i)}$ of p in S_i. Report an L_∞-neighbor that has minimal L_2-distance to p.

LEMMA 7. *The query algorithm reports a $(1+\epsilon)$-approximate L_2-neighbor of p.*

Proof. Consider an exact L_2-neighbor p^* of p. Let i be an index such that the coordinates of $p^* - p$ in $\mathcal{F}_i$ satisfy

$$0 \leq p_{ij} - p^*_{ij} \leq p_{iD} - p^*_{iD} \leq (1+\epsilon)(p_{ij} - p^*_{ij}), \quad 1 \leq j \leq D.$$

Let q be the point that is reported by the algorithm. Since $d_2(p,q) \leq d_2(p,q^{(i)})$, it suffices to prove that $d_2(p,q^{(i)}) \leq (1+\epsilon)d_2(p,p^*)$.

Let B be the D-dimensional box centered at p with sides of length $2d_{i,\infty}(p, p^*)$ that are parallel to the axes of $\mathcal{F}_i$. Since $q^{(i)}$ is an L_∞-neighbor of p in S_i, this point must lie inside or on the boundary of B. It follows that

$$d_2(p, q^{(i)}) \leq \sqrt{D} \cdot d_{i,\infty}(p, p^*).$$

Note that $d_{i,\infty}(p, p^*) = p_{iD} - p^*_{iD}$. Moreover,

$$(d_2(p, p^*))^2 = \sum_{j=1}^{D} (p_{ij} - p^*_{ij})^2 \geq \sum_{j=1}^{D} \left(\frac{p_{iD} - p^*_{iD}}{1+\epsilon} \right)^2 = \frac{D}{(1+\epsilon)^2} (d_{i,\infty}(p, p^*))^2.$$

Hence

$$d_2(p, q^{(i)}) \leq \sqrt{D} \cdot d_{i,\infty}(p, p^*) \leq \sqrt{D} \cdot \frac{1+\epsilon}{\sqrt{D}} \cdot d_2(p, p^*) = (1+\epsilon) d_2(p, p^*).$$

This proves the lemma. □

Hence we have solved the $(1+\epsilon)$-approximate L_2-neighbor problem. The complexity of our solution follows immediately from Theorem 2. We have proved the following theorem.

THEOREM 3. *Let S be a set of n points in $\mathbb{R}^D$ and let ϵ be a positive constant. There exists a data structure of size $O(n(\log n)^{D-2})$ that, given a query point $p \in \mathbb{R}^D$, finds a $(1+\epsilon)$-approximate L_2-neighbor of p in S in $O((\log n)^{D-1})$ time.*

For the dynamic version of the problem, there is a data structure with a query time of $O((\log n)^{D-1} \log\log n)$ and an amortized update time of $O((\log n)^{D-1} \log\log n)$ that uses space $O(n(\log n)^{D-1})$.

In all complexity bounds, the constant factor is proportional to $(c/\epsilon)^{D-1}$ for some fixed c.

5. The containment problem for boxes of constant overlap.

A *box* is a D-dimensional axes-parallel cube, i.e., it is of the form

$$[a_1 : a_1 + \delta] \times [a_2 : a_2 + \delta] \times \cdots \times [a_D : a_D + \delta],$$

for real numbers $a_1, a_2, \ldots, a_D$ and $\delta > 0$. The *center* of this box is the point $(a_1 + \delta/2, a_2 + \delta/2, \ldots, a_D + \delta/2)$.

Let S be a set of n boxes in $\mathbb{R}^D$ that are of *constant overlap*, i.e., there is an integer constant c_D—possibly depending on the dimension—such that each box B of S contains the centers of at most c_D boxes in its interior. (Here we also count the center of B itself. Note that many boxes may have their centers on the boundary of B.)

We want to store these boxes in a data structure such that for any query point $p \in \mathbb{R}^D$, we can find all boxes that contain p. Note that we distinguish between "being contained in a box" and "being contained in the interior of a box." The following lemma shows that a point p can be contained in only a constant number of boxes.

LEMMA 8. *Any point $p \in \mathbb{R}^D$ is contained in at most $2^{1+D^2} c_D$ boxes of S.*

Proof. The proof is by induction on D. So let $D = 1$. Assume w.l.o.g. that $p = 0$. Let S_+ be the set of all boxes of S whose centers are positive. Let $b_1, b_2, \ldots, b_m$ be the centers of the boxes in S_+ that contain p. Assume w.l.o.g. that $b_1 = \max\{b_i : 1 \leq i \leq m\}$. The interior of the box that has b_1 as its center contains all centers b_i, $1 \leq i \leq m$. Hence the constant-overlap property implies that $m \leq c_1$, i.e., point p is contained in at most c_1 boxes of S_+.

By a symmetric argument, point p is contained in at most c_1 boxes of S whose centers are negative. It remains to consider the boxes that have p as their centers. The constant-overlap property directly implies that there are at most c_1 such boxes.

This proves that there are at most $3c_1$ boxes in the entire set S that contain p.

Now let $D \geq 2$ and assume the lemma holds for dimension $D - 1$. Again, we assume w.l.o.g. that p is the origin. Let S_+ be the set of all boxes of S whose centers have positive coordinates. We will show that there are at most c_D boxes in S_+ that contain p.

Let $b_1, b_2, \ldots, b_m$ be the centers of the boxes in S_+ that contain p. Assume w.l.o.g. that

$$\delta := d_\infty(p, b_1) = \max\{d_\infty(p, b_i) : 1 \leq i \leq m\}.$$

Let B be the box of S_+ having b_1 as its center. We claim that $d_\infty(b_1, b_i) < \delta$ for $1 \leq i \leq m$. Indeed, let $1 \leq j \leq D$. The jth coordinate b_{ij} of b_i satisfies $0 < b_{ij} \leq \delta$. As a result, $|b_{1j} - b_{ij}| < \delta$. This proves the claim.

Since B contains p, it has sides of length at least 2δ. Therefore, since each center b_i has L_∞-distance less than δ to b_1, these centers are contained in the interior of B. Then the constant-overlap property implies that $m \leq c_D$.

This proves that p is contained in at most c_D boxes of S_+. By a symmetric argument, p is contained in at most c_D boxes of S that have their centers in a fixed D-dimensional quadrant. Hence p is contained in at most $2^D c_D$ boxes whose centers have nonzero coordinates.

Let S_0 be the set of all boxes in S whose centers have zero as their first coordinate. Consider the set S_0' of $(D - 1)$-dimensional boxes obtained from S_0 by deleting from each box its first coordinates. The constant-overlap property for S_0 implies that the boxes of S_0' are also of constant overlap, with constant c_D. Hence by the induction hypothesis, point $p' = (p_2, \ldots, p_D)$ is contained in at most $2^{1+(D-1)^2} c_D$ boxes of S_0'. These boxes of S_0' correspond exactly to the boxes of S_0 that contain p.

Hence at most $2^{1+(D-1)^2} c_D$ boxes of S_0 contain p. It follows from a symmetric argument that for each $1 \leq i \leq D$, point p is contained in at most $2^{1+(D-1)^2} c_D$ boxes whose centers have zero as their ith coordinate.

To summarize, we have shown that there are at most

$$2^D c_D + D\, 2^{1+(D-1)^2} c_D \leq 2^{1+D^2} c_D$$

boxes of S that contain p. This completes the proof. □

In the §§5.1 and 5.2, we shall give two solutions for the box-containment problem.

5.1. A solution based on segment trees. We start with the one-dimensional case. Let S be a set of n intervals $[a_j : b_j]$, $1 \leq j \leq n$, that are of constant overlap with constant c.

The one-dimensional structure. We store the intervals in the leaves of a balanced binary search tree T, sorted by their right endpoints. The leaves of this tree are threaded in a doubly linked list.

The query algorithm. Let $p \in \mathbb{R}$ be a query element. Search in T for the leftmost leaf containing a right endpoint that is at least equal to p. Starting in this leaf, walk along the leaves to the right and report all intervals encountered that contain p. Stop walking as soon as $4c$ intervals have been reported or when c intervals have been encountered that do not contain p.

LEMMA 9. *The above data structure solves the one-dimensional containment problem in a set of n intervals of constant overlap in $O(n)$ space with a query time of $O(\log n)$. Intervals can be inserted and deleted in $O(\log n)$ time.*

Proof. The complexity bounds are clear; we only have to prove that the query algorithm is correct. Clearly, all intervals that are reported contain p. It remains to show that all intervals containing p are reported.

Let v be the leaf of T in which the search for p ends. The algorithm starts in v and walks to the right. First note that all leaves to the left of v store intervals that do not contain p. We know from Lemma 8 that there are at most $4c$ intervals that contain p. Hence, after $4c$ intervals have been reported, the algorithm can stop. Now assume that in leaf w, we encounter the cth interval that does not contain p. All c encountered intervals that do not contain p lie completely to the right of p. If there is a leaf to the right of w whose interval contains p, then this interval contains these c intervals. By the constant-overlap property, such a leaf cannot exist. This proves that the algorithm can stop in leaf w, i.e., if it has encountered c intervals that do not contain p. □

We next consider the D-dimensional case, where $D \geq 2$. Let S be a set of n boxes in $\mathbb{R}^D$ that are of constant overlap with constant c_D. The data structure is a segment tree for the intervals of the first coordinates of the boxes. (See [13, 15].) The nodes of this tree contain appropriate associated structures.

If $B = B_1 \times B_2 \times \cdots \times B_D$ is a box in $\mathbb{R}^D$, then B' denotes the box $B_2 \times \cdots \times B_D$ in $\mathbb{R}^{D-1}$. Similarly, S' denotes the set $\{B' : B \in S\}$. Recall that we use a similar notation for points.

The D-dimensional structure for $D \geq 2$. Let $a_1 < a_2 < \cdots < a_m$, where $m \leq 2n$, be the sorted sequence of all distinct endpoints of the intervals of the first coordinates of the boxes in S. We store the *elementary intervals*

$$(-\infty : a_1), [a_1 : a_1], (a_1 : a_2), [a_2 : a_2], \ldots, (a_{m-1} : a_m), [a_m : a_m], (a_m : \infty)$$

in this order in the leaves of a balanced binary search tree, called the *main tree*. Each node v of this tree has associated with it an interval I_v that is the union of the elementary intervals of the leaves in the subtree of v. Let S_v be the set of all boxes $B = B_1 \times B_2 \times \cdots \times B_D$ in S such that B_1 spans the interval associated with v but does not span the interval associated with its father node, i.e., $I_v \subseteq B_1$ and $I_{f(v)} \not\subseteq B_1$, where $f(v)$ is the father of v.

Each node v of the main tree contains (a pointer to) an *associated structure* for the set S_v: Partition this set into S_{vl}, S_{vc}, and S_{vr}, consisting of those boxes of S_v whose centers lie to the left of the "vertical" slab $I_v \times \mathbb{R}^{D-1}$, in or on the boundary of this slab, and to the right of this slab, respectively.

The associated structure of v consists of three $(D-1)$-dimensional structures for the sets S'_{vl}, S'_{vc}, and S'_{vr}, that are defined recursively.

Remark. The interval I_v may be open, half-open, or closed. Therefore, the boundary of the slab $I_v \times \mathbb{R}^{D-1}$ does not necessarily belong to this slab.

The query algorithm. Let $p = (p_1, p_2, \ldots, p_D) \in \mathbb{R}^D$ be a query point. Search in the main tree for the elementary interval that contains p_1. For each node v on the search path, recursively perform a $(D-1)$-dimensional query with the point p' in the three structures that are stored with v.

At the last level of the recursion, a one-dimensional query is performed in a binary search tree. In this tree, the algorithm stops if it has reported $2^{1+D^2} c_D$ boxes of S that contain p or if it has encountered c_D boxes of S that do not contain p. (If *in this tree* c_D boxes that do not contain p have been encountered, then the algorithm only stops at *this* level of the recursion. If, on the other hand, an overall $2^{1+D^2} c_D$ boxes that contain p have been reported, then the *entire* query algorithm stops.)

The correctness proof of this algorithm uses the following lemma.

LEMMA 10. *Let x and y be real numbers, let $p \in \mathbb{R}^D$, and let i, $1 \leq i \leq D-1$, be an integer such that $x \leq p_i \leq y$. Let $A = [a_1 : a_1 + \alpha] \times \cdots \times [a_D : a_D + \alpha]$ and*

$B = [b_1 : b_1 + \beta] \times \cdots \times [b_D : b_D + \beta]$ be two boxes such that $a_i \leq x$, $a_i + \alpha \geq y$, $b_i \leq x$, $b_i + \beta \geq y$, $b_D > p_D$, and $b_D + \beta \leq a_D + \alpha$. Finally, assume that p is contained in A. Then

1. *$a_D < b_D + \beta/2 < a_D + \alpha$,*
2. *if one of the following three conditions holds:*
   (a) *$x \leq a_i + \alpha/2 \leq y$ and $x \leq b_i + \beta/2 \leq y$,*
   (b) *$a_i + \alpha/2 < x$ and $b_i + \beta/2 < x$,*
   (c) *$a_i + \alpha/2 > y$ and $b_i + \beta/2 > y$,*

   then $a_i < b_i + \beta/2 < a_i + \alpha$.

Proof. The right inequality of the first assertion follows easily: $b_D + \beta/2 < b_D + \beta \leq a_D + \alpha$. Since p is contained in A, we have $a_D \leq p_D$. Therefore, $a_D \leq p_D < b_D < b_D + \beta/2$. This proves 1.

Before we prove 2, observe that $b_D + \beta \leq a_D + \alpha < b_D + \alpha$. It follows that $\beta < \alpha$.

Assume that case 2(a) applies, i.e., $x \leq a_i + \alpha/2 \leq y$ and $x \leq b_i + \beta/2 \leq y$. Since $b_i + \beta/2 \leq y \leq a_i + \alpha$ and $b_i + \beta/2 \geq x \geq a_i$, we only have to show that $a_i \neq b_i + \beta/2$ and $b_i + \beta/2 \neq a_i + \alpha$.

Assume that $a_i = b_i + \beta/2$. Then $x = a_i = b_i + \beta/2$. First, note that $\alpha/2 \leq y - a_i = y - x$. Since $\beta/2 = b_i + \beta - (b_i + \beta/2) \geq y - (b_i + \beta/2) = y - x$, we infer that $\alpha \leq \beta$. This is a contradiction, and hence $a_i < b_i + \beta/2$.

Next, assume that $b_i + \beta/2 = a_i + \alpha$. Then $y = b_i + \beta/2 = a_i + \alpha$ and, in a similar way, we can prove that $\alpha \leq \beta$. Therefore, $b_i + \beta/2 < a_i + \alpha$.

Next, consider case 2(b), i.e., assume that $a_i + \alpha/2 < x$ and $b_i + \beta/2 < x$. Since $b_i + \beta/2 < x \leq y \leq a_i + \alpha$, we only have to show that $a_i < b_i + \beta/2$. We prove this by contradiction. So assume that $a_i \geq b_i + \beta/2$. Then $\alpha/2 = a_i + \alpha/2 - a_i < x - a_i \leq x - (b_i + \beta/2) = x - b_i - \beta/2$. Since $b_i + \beta \geq y \geq x$, we get $x - b_i \leq \beta$. Therefore, $\alpha/2 < x - b_i - \beta/2 \leq \beta - \beta/2 = \beta/2$, i.e., $\alpha < \beta$. This is a contradiction, and hence we have proved that $a_i < b_i + \beta/2$.

Case 2(c) can be treated in the same way as case 2(b). □

LEMMA 11. *The query algorithm is correct.*

Proof. It is well known that the set of all boxes $B = B_1 \times \cdots \times B_D$ such that $p_1 \in B_1$ is exactly the union of all sets S_v, where v is a node on the search path to p_1. Hence we only have to consider the nodes on this search path.

Let v be a node on the path to p_1. We know that p_1 is contained in the first interval of each box of S_v. Hence we have to find all boxes in S_v whose last $D - 1$ intervals contain $(p_2, \ldots, p_D)$. We claim that the recursive queries in the three structures stored with v find these boxes. By Lemma 8, there are at most $2^{1+D^2} c_D$ boxes that contain p. Hence, at the last level of the recursion, the query algorithm can stop as soon as it has reported this many boxes. It remains to prove that, at the last level, the query algorithm can stop if it has encountered c_D boxes that do not contain p.

Consider such a last level. That is, let $v_1, v_2, \ldots, v_{D-1}$ be nodes such that

1. $v_1 = v$,
2. v_i is a node of the main tree of one of the three structures that are stored with v_{i-1},
3. v_i lies on the search path to p_i.

The algorithm makes one-dimensional queries with p_D in the three structures—binary search trees—that are stored with v_{D-1}.

Consider one such query. The algorithm searches for p_D. Let r be the leaf in which this search ends. Starting in r, the algorithm walks along the leaves to the right. During this walk, it encounters boxes that do or do not contain p. Assume that in leaf s, the c_Dth box is encountered that does not contain p. We have to show that the algorithm can stop in s. That is, we must show that all leaves to the right of s store boxes that do not contain p.

Assume this is not the case. Then there is a box

$$A = [a_1 : a_1 + \alpha] \times [a_2 : a_2 + \alpha] \times \cdots \times [a_D : a_D + \alpha]$$

that is stored in a leaf to the right of s and that contains p. Let

$$B^{(j)} = [b_{j1} : b_{j1} + \beta_j] \times [b_{j2} : b_{j2} + \beta_j] \times \cdots \times [b_{jD} : b_{jD} + \beta_j], 1 \leq j \leq c_D,$$

be the encountered boxes between r and s that do not contain p. We shall prove that A contains the centers of all these boxes in its interior. This will be a contradiction.

Let $1 \leq j \leq c_D$. Since the leaf of A lies to the right of the leaf of $B^{(j)}$, we know that $b_{jD} + \beta_j \leq a_D + \alpha$.

Let $1 \leq i \leq D - 1$ and let $x \leq y$ be the boundary points of the interval I_{v_i}. Then $x \leq p_i \leq y$ because $p_i \in I_{v_i}$. Moreover, we know that I_{v_i} is contained in the ith interval of A and $B^{(j)}$, i.e., $a_i \leq x$, $a_i + \alpha \geq y$, $b_{ji} \leq x$ and $b_{ji} + \beta_j \geq y$. Finally, since the leaf of $B^{(j)}$ is equal to leaf r or lies to the right of it, we know that $p_D \leq b_{jD} + \beta_j$. Since $B^{(j)}$ does not contain p, this implies that $b_{jD} > p_D$. (Note that the ith interval of $B^{(j)}$ contains p_i, $1 \leq i \leq D - 1$.)

The definition of our data structure implies that the ith coordinates of the centers of A and $B^{(j)}$ are either both on the boundary or contained in I_{v_i}, or both are less than x, or both are larger than y. Therefore, all requirements of Lemma 10 are satisfied, and we conclude that $a_i < b_{ji} + \beta_j/2 < a_i + \alpha$.

Since i was arbitrary between 1 and $D - 1$, and since Lemma 10 also implies that $a_D < b_{jD} + \beta_j/2 < a_D + \alpha$, we have proved that the center of $B^{(j)}$ is contained in the interior of A. This completes the proof. □

We analyze the complexity of the D-dimensional structure. Let $M(n, D)$ denote the size of the data structure. It is well known that the first interval of a box in S is stored in the associated structure of $O(\log n)$ nodes of the main tree. This implies that $M(n, D) = O(M(n, D - 1) \log n)$. Since $M(n, 1) = O(n)$, we get $M(n, D) = O(n(\log n)^{D-1})$. Using presorting, the structure can be built in $O(n(\log n)^{D-1})$ time.

Let $Q(n, D)$ denote the query time. Then $Q(n, 1) = O(\log n)$. The query algorithm performs $(D - 1)$-dimensional queries in each of the $O(\log n)$ nodes on the search path. As a result, the query time satisfies $Q(n, D) = O(Q(n, D-1) \log n)$, which solves to $O((\log n)^D)$. By applying fractional cascading (see [6]) in the two-dimensional case, we decrease $Q(n, 2)$ to $O(\log n)$. This improves the query time to $Q(n, D) = O((\log n)^{D-1})$ for $D \geq 2$.

By applying standard techniques, the data structure can be adapted to handle insertions and deletions of boxes. (See [13, 14].) Because of dynamic fractional cascading, the query time increases by a factor of $O(\log \log n)$, whereas the size only increases by a constant factor. The amortized update time is $O((\log n)^{D-1} \log \log n)$.

We summarize our result.

THEOREM 4. *Let S be a set of n boxes in $\mathbb{R}^D$ of constant overlap. There exists a data structure of size $O(n(\log n)^{D-1})$ such that for any point $p \in \mathbb{R}^D$, we can find all boxes of S that contain p in $O((\log n)^{D-1})$ time. This static data structure can be built in $O(n(\log n)^{D-1})$ time.*

For the dynamic version of the problem, the query time is $O((\log n)^{D-1} \log \log n)$ and the size of the structure is $O(n(\log n)^{D-1})$. Boxes can be inserted and deleted in $O((\log n)^{D-1} \log \log n)$ amortized time.

5.2. A solution based on skewer trees. In this section, we give a linear-space solution to the box-containment problem. This solution is based on skewer trees, introduced by Edelsbrunner et al. [8]. (See also Smid [21] for a dynamic version of this data structure.)

Let S be a set of n boxes in $\mathbb{R}^D$ that are of constant overlap with constant c_D. For $D = 1$, the data structure is the same as in the previous subsection.

The D-dimensional structure for $D \geq 2$. If S is empty, then the data structure is also empty. Assume that S is nonempty. Let γ be the median of the set

$$\{(a_1 + b_1)/2 : [a_1 : b_1] \times [a_2 : b_2] \times \cdots \times [a_D : b_D] \in S\},$$

and let ϵ be the largest element that is less than γ in the set of all elements a_1, $(a_1 + b_1)/2$, and b_1, where $[a_1 : b_1] \times \cdots \times [a_D : b_D]$ ranges over S. Let $\gamma_1 := \gamma - \epsilon/2$ and let σ be the hyperplane in $\mathbb{R}^D$ with equation $x_1 = \gamma_1$.

Let $S_<$, S_0, and $S_>$ be the set of boxes $[a_1 : b_1] \times \cdots \times [a_D : b_D]$ in S such that $b_1 < \gamma_1$, $a_1 \leq \gamma_1 \leq b_1$, and $\gamma_1 < a_1$, respectively. The D-dimensional data structure for the set S is an augmented binary search tree—called the *main tree*—having the following form:

1. The root contains the hyperplane σ.
2. The root contains pointers to its left and right sons, which are D-dimensional structures for the sets $S_<$ and $S_>$, respectively.
3. The root contains (a pointer to) an *associated structure* for the set S_0: Partition this set into S_{0l} and S_{0r}, consisting of those boxes in S_0 whose centers lie to the left of the hyperplane σ and to the right of σ, respectively.
   The associated structure of the root consists of two $(D-1)$-dimensional structures for the sets S'_{0l} and S'_{0r}.

Remark. By our choice of γ_1, the hyperplane σ does not contain the center of any box in S. Moreover, each of the sets $S_<$ and $S_>$ has size at most $n/2$. The set S_0 may have size n. The *height* of the data structure is defined as the height of its main tree. It follows that the structure has height $O(\log n)$.

The query algorithm. Let $p = (p_1, p_2, \ldots, p_D) \in \mathbb{R}^D$ be a query point. Let $\sigma : x_1 = \gamma_1$ be the hyperplane stored in the root of the main tree. Recursively perform a $(D-1)$-dimensional query with the point p' in the two structures that are stored with the root.

If $p_1 < \gamma_1$ (resp. $p_1 > \gamma_1$), then recursively perform a D-dimensional query with p in the left (resp. right) subtree of the root, unless this subtree is empty, in which case the algorithm stops.

At the last level of the recursion, a one-dimensional query is performed in a binary search tree. In this tree, the algorithm stops if it has reported $2^{1+D^2} c_D$ boxes of S that contain p, or if it has encountered c_D boxes of S whose last intervals do not contain p_D, or if it has encountered $2^{1+D^2} c_D$ boxes of S that do not contain p but whose last intervals contain p_D.

The correctness proof is similar to the that of §5.1. Again, we start with a technical lemma.

LEMMA 12. *Let $p \in \mathbb{R}^D$ and let $\sigma_i : x_i = \gamma_i$, $1 \leq i \leq D-1$, be hyperplanes in $\mathbb{R}^D$. Let $A = [a_1 : a_1 + \alpha] \times \cdots \times [a_D : a_D + \alpha]$ and $B = [b_1 : b_1 + \beta] \times \cdots \times [b_D : b_D + \beta]$ be two boxes such that $b_D > p_D$ and $b_D + \beta \leq a_D + \alpha$. Assume that p is contained in A. Finally, assume that $a_i + \alpha/2 < \gamma_i \leq a_i + \alpha$ and $b_i + \beta/2 < \gamma_i \leq b_i + \beta$ for all $1 \leq i \leq D-1$.*

Then $a_i < b_i + \beta/2 < a_i + \alpha$ for all $1 \leq i \leq D$.

Proof. We start with $i = D$. It follows directly that $b_D + \beta/2 < b_D + \beta \leq a_D + \alpha$. Since p is contained in A, we have $a_D \leq p_D$. Therefore, $a_D \leq p_D < b_D < b_D + \beta/2$.

Let $1 \leq i \leq D-1$. Then $b_i + \beta/2 < \gamma_i \leq a_i + \alpha$. It remains to show that $a_i < b_i + \beta/2$. Assume that $b_i + \beta/2 \leq a_i$. Then, $\beta/2 = b_i + \beta - (b_i + \beta/2) \geq \gamma_i - (b_i + \beta/2) \geq \gamma_i - a_i > \alpha/2$, i.e., $\beta > \alpha$. But since $b_D + \beta \leq a_D + \alpha < b_D + \alpha$, we also have $\beta < \alpha$. This is a contradiction. □

LEMMA 13. *The query algorithm is correct.*

Proof. It is clear that the algorithm branches correctly. Hence we only have to consider the last level of the recursion. Since there are at most $2^{1+D^2} c_D$ boxes that contain p, the algorithm can stop as soon as it has reported this many boxes. It remains to prove that, at the last level, the query algorithm can stop if it has encountered c_D boxes whose last intervals do not contain p_D or if it has encountered $2^{1+D^2}c_D$ boxes that do not contain p but whose last intervals contain p_D.

Consider such a last level. That is, let $v_1, v_2, \ldots, v_{D-1}$ be nodes such that

1. v_1 is a node of the main tree,
2. v_i is a node of the main tree of one of the two structures that are stored with v_{i-1},
3. v_i lies on the search path to p_i.

Let $\sigma_i : x_i = \gamma_i$ be the hyperplane that is stored with v_i, $1 \leq i \leq D-1$.

The algorithm makes one-dimensional queries with p_D in the two structures—binary trees—that are stored with v_{D-1}. Consider one such query. The algorithm searches for p_D. Let r be the leaf in which this search ends. Starting in r, the algorithm walks along the leaves to the right. Assume that in leaf s, the c_Dth box is encountered whose last interval does not contain p_D, or the $(2^{1+D^2}c_D)$th box is encountered that does not contain p but whose last interval contains p_D. We will prove that all leaves to the right of s store boxes that do not contain p.

Assume this is not the case. Then there is a box

$$A = [a_1 : a_1 + \alpha] \times [a_2 : a_2 + \alpha] \times \cdots \times [a_D : a_D + \alpha]$$

that is stored in a leaf to the right of s and that contains p. Let

$$B^{(j)} = [b_{j1} : b_{j1} + \beta_j] \times [b_{j2} : b_{j2} + \beta_j] \times \cdots \times [b_{jD} : b_{jD} + \beta_j]$$

be the encountered boxes between r and s that do not contain p.

The definition of our data structure implies that for each $1 \leq i \leq D-1$, the centers of A and the $B^{(j)}$'s lie on the same side of the hyperplane σ_i. Moreover, these boxes intersect σ_i. We assume w.l.o.g. that for all $1 \leq i \leq D-1$, $a_i + \alpha/2 < \gamma_i \leq a_i + \alpha$, and for all j and all $1 \leq i \leq D-1$, $b_{ji} + \beta_j/2 < \gamma_i \leq b_{ji} + \beta_j$.

There are two possible cases. First, assume that the last intervals of c_D boxes $B^{(j)}$ do not contain p_D. Consider such a box $B^{(j)}$. Since the leaf of this box is equal to leaf r or lies to the right of it, we must have $p_D \leq b_{jD} + \beta_j$. Hence $p_D < b_{jD}$. Also, since the leaf of A lies to the right of the leaf of $B^{(j)}$, we have $b_{jD} + \beta_j \leq a_D + \alpha$. Hence all requirements of Lemma 12 are satisfied. We conclude that the center of $B^{(j)}$ is contained in the interior of A. This proves that the interior of A contains more than c_D centers, namely its own center and the centers of c_D boxes $B^{(j)}$. This is a contradiction.

The second case is where $2^{1+D^2}c_D$ boxes $B^{(j)}$ do not contain p but their last intervals contain p_D. We claim that the point $q := (\gamma_1, \gamma_2, \ldots, \gamma_{D-1}, p_D)$ is contained in A and in all these $B^{(j)}$'s. To prove this, first note that $a_D \leq p_D \leq a_D + \alpha$ because A contains p. Consider any of these boxes $B^{(j)}$. By our assumption, $b_{jD} \leq p_D \leq b_{jD} + \beta_j$. Our assumptions also imply that $a_i \leq \gamma_i \leq a_i + \alpha$ and $b_{ji} \leq \gamma_i \leq b_{ji} + \beta_j$ for all $1 \leq i \leq D-1$. Hence there are more than $2^{1+D^2}c_D$ boxes that contain q. This contradicts Lemma 8. □

The complexity analysis is similar to that of §5.1. Since the main tree has height $O(\log n)$, the query time is bounded by $O((\log n)^D)$. In the planar case, we can apply fractional cascading to improve the query time to $O(\log n)$. Then the query time for the D-dimensional case is improved to $O((\log n)^{D-1})$. It is easy to see that the data structure has size $O(n)$ and that it can be built in $O(n \log n)$ time. (See [8] for details.)

We can adapt the data structure such that it can also handle insertions and deletions of boxes. Since the algorithms and their running times are exactly the same as in [21], we refer the reader to that paper for the details. Because of dynamic fractional cascading, the query time increases by a factor of $O(\log\log n)$, whereas the size only increases by a constant factor. The amortized update time is $O((\log n)^2 \log\log n)$.

We summarize our results.

THEOREM 5. *Let S be a set of n boxes in $\mathbb{R}^D$ of constant overlap. There exists a data structure of size $O(n)$ such that for any point $p \in \mathbb{R}^D$, we can find all boxes of S that contain p in $O((\log n)^{D-1})$ time. This static data structure can be built in $O(n \log n)$ time.*

For the dynamic version of the problem, the query time is $O((\log n)^{D-1} \log\log n)$ and the size of the structure is $O(n)$. Boxes can be inserted and deleted in amortized time $O((\log n)^2 \log\log n)$.

6. Maintaining the closest pair. In this section, we apply the results obtained so far to maintain a closest pair of a point set under insertions and deletions. Let S be a set of n points in $\mathbb{R}^D$ and let $1 \leq t \leq \infty$ be a real number. We denote the L_t-distance between any two points p and q in $\mathbb{R}^D$ by $d(p, q)$. The pair P, $Q \in S$ is called a *closest pair* of S if

$$d(P, Q) = \min\{d(p, q) : p, q \in S, p \neq q\}.$$

We introduce the following notations. For any point $p \in \mathbb{R}^D$, box(p) denotes the smallest box centered at p that contains at least $(2D + 2)^D$ points of $S \setminus \{p\}$. In other words, the side length of box(p) is twice the L_∞-distance between p and its $(1 + (2D + 2)^D)$th (resp. $(2D + 2)^D$th) L_∞-neighbor, if $p \in S$ (resp. $p \notin S$).

Let $N(p)$ be the set of points of $S \setminus \{p\}$ that are contained in the interior of box(p). Note that $N(p)$ has size less than $(2D + 2)^D$. In fact, $N(p)$ may even be empty.

Our data structure is based on the following lemma.

LEMMA 14. *The set $\{(p, q) : p \in S, q \in N(p)\}$ contains a closest pair of S.*

Proof. Let (P, Q) be a closest pair of S. We have to show that $Q \in N(P)$. Assume this is not the case. Let δ be the side length of box(P). Since Q lies outside or on the boundary of this box, we have $d(P, Q) \geq \delta/2$.

Partition box(P) into $(2D+2)^D$ subboxes with sides of length $\delta/(2D+2)$. Since box(P) contains at least $1 + (2D+2)^D$ points of S, one of these subboxes contains at least two points. These two points have distance at most $D \cdot \delta/(2D+2) < \delta/2$, which is a contradiction because (P, Q) is a closest pair of S. □

The set $\{\text{box}(p) : p \in S\}$ is of constant overlap: each box contains the centers of at most $(2D + 2)^D$ boxes in its interior. These centers are precisely the points of $N(p) \cup \{p\}$. This fact and the above lemma suggest the following data structure.

The closest-pair data structure.

1. The points of S are stored in a range tree.
2. The distances of the multiset $\{d(p, q) : p \in S, q \in N(p)\}$ are stored in a heap. With each distance, we store the corresponding pair of points. (Note that both $d(p, q)$ and $d(q, p)$ may occur in the heap.)
3. The points of S are stored in a dictionary. With each point p, we store a list containing the elements of $N(p)$. For convenience, we also call this list $N(p)$. With each point q in $N(p)$, we store a pointer to the occurrence of $d(p, q)$ in the heap.
4. The set $\{\text{box}(p) : p \in S\}$ is stored in the dynamic data structure of Theorem 4. This structure is called the *box tree*.

It follows from Lemma 14 that the pair of points that is stored with the minimal element of the heap is a closest pair of S. The update algorithms are rather straightforward.

The insertion algorithm. Let $p \in \mathbb{R}^D$ be the point to be inserted. Assume w.l.o.g. that $p \notin S$.

1. Using the range tree, find the $(2D + 2)^D$ L_∞-neighbors of p in S. The point among these neighbors that has maximal L_∞-distance to p determines box(p). The neighbors that are contained in the interior of box(p) form the list $N(p)$.

2. Insert p into the range tree and insert the distances $d(p, q)$, $q \in N(p)$ into the heap. Then, insert p—together with the list $N(p)$—into the dictionary. With each point q in $N(p)$, store a pointer to $d(p, q)$ in the heap. Finally, insert box(p) into the box tree.
3. Using the box tree, find all boxes that contain p. For each reported element box(q), $q \neq p$, that contains p in its interior, do the following:
   (a) Search in the dictionary for q. Insert p into $N(q)$, insert $d(q, p)$ into the heap, and store with p a pointer to $d(q, p)$.
   (b) If $N(q)$ has size less than $(2D+2)^D$, then the insertion algorithm is completed. Otherwise, if $N(q)$ has size $(2D+2)^D$, let $r_1, \dots, r_l$ be all points in $N(q)$ that have maximal L_∞-distance to q. For each $1 \leq i \leq l$, delete r_i from $N(q)$ and delete $d(q, r_i)$ from the heap. Finally, delete box(q) from the box tree and insert the box centered at q that has r_1 on its boundary as the new box(q).

The deletion algorithm. Let $p \in S$ be the point to be deleted.

1. Delete p from the range tree. Search for p in the dictionary. For each point q in $N(p)$, delete the distance $d(p, q)$ from the heap. Then delete p and $N(p)$ from the dictionary. Finally, delete box(p) from the box tree.
2. Using the box tree, find all boxes that contain p. For each reported element box(q), do the following:
   (a) If p lies in the interior of box(q), then search in the dictionary for q, delete p from $N(q)$, and delete $d(q, p)$ from the heap.
   (b) Using the range tree, find the $1 + (2D+2)^D$ L_∞-neighbors of q. Let box$_0(q)$ be the smallest box centered at q that contains these neighbors. If box$(q) =$ box$_0(q)$, then the deletion algorithm is completed.
   (c) Otherwise, if box$(q) \neq$ box$_0(q)$, let $r_1, \dots, r_l$ be all points that are contained in the interior of box$_0(q)$ but that do not belong to $N(q) \cup \{q\}$. For each $1 \leq i \leq l$, insert r_i into $N(q)$, insert $d(q, r_i)$ into the heap, and store with r_i a pointer to $d(q, r_i)$. Finally, delete box(q) from the box tree and insert box$_0(q)$, which is the new box(q).

It is easy to verify that these update algorithms correctly maintain the closest-pair data structure. During these algorithms, we perform a constant number of query and update operations in the range tree, the box tree, the heap, and the dictionary. Therefore, by Theorems 2 and 4, the amortized update time of the entire data structure is bounded by $O((\log n)^{D-1} \log\log n)$. We have proved the following result.

THEOREM 6. *Let S be a set of n points in $\mathbb{R}^D$ and let $1 \leq t \leq \infty$. There exists a data structure of size $O(n(\log n)^{D-1})$ that maintains an L_t-closest pair of S in $O((\log n)^{D-1} \log\log n)$ amortized time per insertion and deletion. The constant factor in the space (resp. update time) bound is of the form $O(D)^D$ (resp. $2^{O(D^2)}$).*

Consider our data structure again. The box box(p) that is associated with point p contains an L_∞-neighbor of p in $S \setminus \{p\}$. Therefore, the data structure can easily be adapted such that it maintains for each point in S its L_∞-neighbor.

COROLLARY 1. *Let S be a set of n points in $\mathbb{R}^D$. There exists a data structure of size $O(n(\log n)^{D-1})$ that maintains an L_∞-neighbor of each point in S. This data structure has an amortized update time of $O((\log n)^{D-1} \log\log n)$.*

7. A transformation for reducing the space complexity. The closest-pair data structure of §6 uses more than linear space. This raises the question of whether the same update time can be obtained using only linear space. In this section, we show that for $D \geq 3$, this is indeed possible. For $D = 2$, we will obtain a family of closest-pair data structures.

Note that we have a linear-space solution for maintaining the set $\{\text{box}(p) : p \in S\}$. (See Theorem 5.) For the L_∞-neighbor problem, however, no linear-space solution that has polylogarithmic query and update times is known. Hence, in order to reduce the space complexity, we should avoid using the range tree.

We will give a transformation that, given *any* dynamic closest-pair data structure having more than linear size, produces another dynamic closest-pair structure that uses less space.

The transformed data structure is composed on two sets A and B that partition S. The set B is contained in a dynamic data structure. To reduce space, B is a subset of the entire set and contains points involved in $o(n)$ updates only.

Let DS be a data structure that maintains a closest pair in a set of n points in $\mathbb{R}^D$ under insertions and deletions. Let $S(n)$ and $U(n)$ denote the size and update times of DS, respectively. The update time may be worst-case or amortized. We assume that $S(n)$ and $U(n)$ are nondecreasing and *smooth* in the sense that $S(\Theta(n)) = \Theta(S(n))$ and $U(\Theta(n)) = \Theta(U(n))$. Finally, let $f(n)$ be a nondecreasing smooth integer function such that $1 \leq f(n) \leq n/2$.

Let $S \subseteq \mathbb{R}^D$ be the current set of points. The cardinality of S is denoted by n. Our transformed data structure will be completely rebuilt after a sufficiently long sequence of updates. Let S_0 be the set of points at the moment of the most recent rebuilding and let n_0 be its size at that moment.

As in the previous section, for each $p \in \mathbb{R}^D$, $\text{box}(p)$ denotes the smallest box centered at p that contains at least $(2D+2)^D$ points of $S \setminus \{p\}$. The set of all points of $S \setminus \{p\}$ that are in the interior of this box is denoted by $N(p)$. If $p \in S_0$, then $\text{box}_0(p)$ denotes the smallest box centered at p that contains at least $(2D+2)^D$ points of $S_0 \setminus \{p\}$.

The transformed closest-pair data structure.

1. The set S is partitioned into sets A and B such that $A \subseteq \{p \in S : p \in S_0 \wedge \text{box}(p) \subseteq \text{box}_0(p)\}$.
2. The distances of the multiset $\{d(p,q) : p \in A, q \in N(p)\}$ are stored in a heap. With each distance, we store the corresponding pair of points.
3. The boxes of the set $\{\text{box}_0(p) : p \in S_0\}$ are stored in a list called the *box list*. With each element $\text{box}_0(p)$ in this list, we store a bit that has value *true* if and only if $p \in A$. Moreover, if $p \in A$, we store with $\text{box}_0(p)$ the box $\text{box}(p)$.
4. The boxes of the set $\{\text{box}_0(p) : p \in S_0\}$ are stored in the *static* data structure of Theorem 5. This structure is called the *box tree*. With each box in this structure, we store a pointer to its occurrence in the box list.
5. The points of S are stored in a dictionary. With each point p, we store a bit that indicates whether p belongs to A or B. If $p \in A$, then we store with p
   (a) a pointer to the occurrence of $\text{box}_0(p)$ in the box list, and
   (b) a list containing the elements of $N(p)$. For convenience, we also call this list $N(p)$. With each point q in $N(p)$, we store a pointer to the occurrence of $d(p,q)$ in the heap.
6. The set B is stored in the dynamic data structure DS. This structure is called the *B-structure*.

First we prove that this data structure indeed enables us to find a closest pair of the current set S in $O(1)$ time.

LEMMA 15. *Let δ be the minimal distance stored in the heap and let δ' be the distance of a closest pair in B. Then,* $\min(\delta, \delta')$ *is the distance of a closest pair in the set S.*

Proof. Let (P, Q) be a closest pair in S. We distinguish two cases.

Case 1. At least one of P and Q is contained in A.

Assume w.l.o.g. that $P \in A$. Since $\text{box}(P)$ contains at least $1 + (2D+2)^D$ points of S, it follows in the same way as in the proof of Lemma 14 that Q is contained in the interior of this box. Hence $Q \in N(P)$ and, therefore, the distance $d(P,Q)$ is stored in the heap. Clearly, the heap contains only distances of the current set S. Therefore, $\delta = d(P,Q)$.

Moreover, since $d(P, Q)$ is the minimal distance in S, we have $d(P, Q) \leq \delta'$. This proves that $d(P, Q) = \min(\delta, \delta')$.

Case 2. Both P and Q are contained in B.

Since $d(P, Q)$ is the minimal distance in S and since the heap only stores distances between points of the current set S, we have $\delta' = d(P, Q)$ and $d(P, Q) \leq \delta$. Therefore, $d(P, Q) = \min(\delta, \delta')$. □

Initialization. At the moment of initialization, $S = S_0 = A$ and $B = \emptyset$. Using Vaidya's algorithm [24], compute for each point p in S its $1 + (2D + 2)^D$ L_∞-neighbors. The point among these neighbors that has maximal L_∞-distance to p determines $\text{box}(p) = \text{box}_0(p)$. The neighbors (except p itself) that are contained in the interior of this box form the list $N(p)$. It is clear how the rest of the data structure can be built. Note that each element $\text{box}_0(p)$ in the box list has a bit with value *true*.

Now we can give the update algorithms. If a point p of S_0 is deleted, it may be inserted again during some later update operation. If this happens, p is assumed to be a new point, i.e., it is assumed that p does not belong to S_0 again. In this way, an inserted point always belongs to B.

The insertion algorithm. Let $p \in \mathbb{R}^D$ be the point to be inserted. Assume w.l.o.g. that $p \notin S$.

1. Insert p into the dictionary and store with p a bit that says that p belongs to B. Then insert p into the B-structure.
2. Using the box tree, find all boxes that contain p. For each reported element $\text{box}_0(q)$, follow the pointer to its occurrence in the box list. If the bit of $\text{box}_0(q)$ has value *true*, then check if p is contained in the interior of $\text{box}(q)$. If so, do the following:
   (a) Search in the dictionary for q. Insert p into $N(q)$, insert $d(q, p)$ into the heap, and store with p a pointer to $d(q, p)$.
   (b) If $N(q)$ has size less than $(2D+2)^D$, then the insertion algorithm is completed. Otherwise, let $r_1, \ldots, r_l$ be all points of $N(q)$ that are at maximal L_∞-distance from q. For each $1 \leq i \leq l$, delete r_i from $N(q)$ and delete $d(q, r_i)$ from the heap. Finally, replace $\text{box}(q)$—which is stored with $\text{box}_0(q)$ in the box list—by the box centered at q that has r_1 on its boundary; this is the new $\text{box}(q)$.

It is easy to verify that this algorithm correctly maintains the data structure. Note that since $A \subseteq \{p \in S : p \in S_0 \wedge \text{box}(p) \subseteq \text{box}_0(p)\}$, all boxes $\text{box}(q)$, $q \in A$, that contain p are found in step 2.

The deletion algorithm. Let $p \in S$ be the point to be deleted.

1. Search for p in the dictionary. If $p \in B$, then delete p from this dictionary and from the B-structure.
   Otherwise, if $p \in A$, follow the pointer to $\text{box}_0(p)$ in the box list and set its bit to *false*. Moreover, for each point q in $N(p)$, delete the distance $d(p, q)$ from the heap. Then delete p from the dictionary.
2. Using the box tree, find all boxes that contain p. For each reported element $\text{box}_0(q)$, follow the pointer to its occurrence in the box list. If the bit of $\text{box}_0(q)$ has value *true*, then check if p is contained in $\text{box}(q)$. If so, do the following:
   Set the bit of $\text{box}_0(q)$ to *false*. Search in the dictionary for q. For each point r in $N(q)$, delete $d(q, r)$ from the heap. Then delete the pointer from q to $\text{box}_0(q)$, delete the list $N(q)$, and store with q a bit saying that it belongs to B. Finally, insert q into the B-structure.

This concludes the description of the update algorithms. In order to guarantee a good space bound, we occasionally rebuild the data structure as follows.

Rebuild. Recall that n_0 is the size of S at the moment we initialize the structure. After $f(n_0)$ updates have been performed, we discard the entire structure and initialize a new data structure for the current S.

We analyze the complexity of the transformed data structure. First, note that the initial and current sizes n_0 and n are proportional: Since $f(n_0) \leq n_0/2$, we have $n \leq n_0 + f(n_0) \leq 3n_0/2$ and $n \geq n_0 - f(n_0) \geq n_0/2$.

The total size of the heap, the box list, the box tree, and the dictionary is bounded by $O(n + n_0) = O(n)$. Consider the B-structure. Initially, this structure is empty. With each insertion, we insert one point into it, whereas with each deletion, at most a constant number of points are inserted. (See Lemma 8 for the constant. Note that $c_D = (2D+2)^D$.) Therefore, the B-structure stores $O(f(n_0)) = O(f(n))$ points and it has size $O(S(f(n)))$.

During each update operation, we perform a constant number of queries and updates in the various parts of the structure. Therefore, by Theorem 5, $O((\log n)^{D-1} + U(f(n)))$ time is spent per update, in case the data structure is not rebuilt.

Consider the initialization. We can compute each point's $1 + (2D+2)^D$ L_∞-neighbors in $O(n_0 \log n_0)$ time. (See [24].) By Theorem 5, the static box tree can also be built in $O(n_0 \log n_0)$ time. It is clear that the rest of the data structure can be built within these time bounds. Hence the entire initialization takes $O(n_0 \log n_0)$ time. Since we do not rebuild during the next $f(n_0)$ updates, the initialization adds

$$O\left(\frac{n_0 \log n_0}{f(n_0)}\right) = O((n \log n)/f(n))$$

to the overall amortized update time. This proves the following result.

THEOREM 7. *Let* DS *be any data structure for the dynamic closest-pair problem. Let $S(n)$ and $U(n)$ denote the size and update time of* DS, *respectively. Let* $1 \leq f(n) \leq n/2$ *be a nondecreasing integer function. Assume that $S(n)$, $U(n)$, and $f(n)$ are smooth.*

We can transform DS *into another data structure for the dynamic closest-pair problem; it has*

1. *size $O(n + S(f(n)))$, and*
2. *an amortized update time of $O((\log n)^{D-1} + U(f(n)) + (n \log n)/f(n))$.*

COROLLARY 2. *Let S be a set of n points in $\mathbb{R}^D$, $D \geq 3$, and let $1 \leq t \leq \infty$. There exists a data structure of size $O(n)$ that maintains an L_t-closest pair of S in $O((\log n)^{D-1} \log\log n)$ amortized time per insertion and deletion.*

Proof. We apply Theorem 7 twice. Let DS be the data structure of Theorem 6, i.e., $S(n) = O(n(\log n)^{D-1})$ and $U(n) = O((\log n)^{D-1} \log\log n)$. Moreover, let $f(n) = n/((\log n)^{D-2} \log\log n)$. Then Theorem 7 gives a closest-pair structure DS′ of size $S'(n) = O(n \log n/\log\log n)$ that has $U'(n) = O((\log n)^{D-1} \log\log n)$ amortized update time. Applying Theorem 7 to DS′ with $f'(n) = n \log\log n/\log n$ proves the corollary. □

COROLLARY 3. *Let S be a set of n points in the plane and let $1 \leq t \leq \infty$. For any nonnegative integer constant k, there exists a data structure*

1. *of size $O(n \log n/(\log\log n)^k)$ that maintains an L_t-closest pair of S at a cost of $O(\log n \log\log n)$ amortized time per insertion and deletion,*
2. *of size $O(n)$ that maintains an L_t-closest pair of S in $O((\log n)^2/(\log\log n)^k)$ amortized time per insertion and deletion.*

Proof. The proof is by induction on k. For $k = 0$, the result claimed in 1 follows from Theorem 6. Let $k \geq 0$ and let DS be a closest-pair data structure that has size $S(n) = O(n \log n/(\log\log n)^k)$ and update time $U(n) = O(\log n \log\log n)$. Applying Theorem 7 with $f(n) = n(\log\log n)^k/\log n$ gives a closest-pair structure of size $O(n)$ that has $O((\log n)^2/(\log\log n)^k)$ amortized update time.

On the other hand, applying Theorem 7 to DS with $f(n) = n/\log\log n$ gives a closest-pair structure of size $O(n\log n/(\log\log n)^{k+1})$ with an amortized update time of $O(\log n \log\log n)$. $\square$

8. Concluding remarks. We have given new techniques for solving the dynamic approximate nearest-neighbor problem and the dynamic closest-pair problem. Note that for the static version of the first problem, an optimal solution—with logarithmic query time and linear size—is known. (See [2].) It would be interesting to solve the dynamic problem within the same complexity bounds and with a logarithmic update time.

For the dynamic closest-pair problem, we obtained several new results. We first gave a data structure that improved the best structures that were known. Then we applied a general transformation to improve this solution even further. Note that if we apply this transformation several times, as we did, then we maintain a hierarchy of data structures similar to the logarithmic method for decomposable searching problems. (See [3].) It would be interesting to know if the ideas of this transformation can be applied to other problems. Finally, we leave as an open problem to decide whether there is an $O(n)$-space data structure that maintains the closest pair in $O(\log n)$ time per insertion and deletion.

REFERENCES

[1] S. Arya and D. M. Mount, *Approximate nearest neighbor queries in fixed dimensions*, in Proc. 4th Annual ACM–SIAM Symposium on Discrete Algorithms, Society for Industrial and Applied Mathematics, Philadelphia, 1993, pp. 271–280.

[2] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Wu, *An optimal algorithm for approximate nearest neighbor searching*, in Proc. 5th Annual ACM–SIAM Symposium on Discrete Algorithms, Society for Industrial and Applied Mathematics, Philadelphia, 1994, pp. 573–582.

[3] J. L. Bentley, *Decomposable searching problems*, Inform. Process. Lett., 8 (1979), pp. 244–251.

[4] J. L. Bentley and M. I. Shamos, *Divide-and-conquer in multidimensional space*, in Proc. 8th Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1976, pp. 220–230.

[5] M. Bern, *Approximate closest-point queries in high dimensions*, Inform. Process. Lett., 45 (1993), pp. 95–99.

[6] B. Chazelle and L. J. Guibas, *Fractional cascading* I: *A data structuring technique*, Algorithmica, 1 (1986), pp. 133–162.

[7] K. L. Clarkson, *A randomized algorithm for closest-point queries*, SIAM J. Comput., 17 (1988), pp. 830–847.

[8] H. Edelsbrunner, G. Haring, and D. Hilbert, *Rectangular point location in d dimensions with applications*, Comput. J., 29 (1986), pp. 76–82.

[9] M. Golin, R. Raman, C. Schwarz, and M. Smid, *Randomized data structures for the dynamic closest-pair problem*, in Proc. 4th Annual ACM–SIAM Symposium on Discrete Algorithms, Society for Industrial and Applied Mathematics, Philadelphia, 1993, pp. 301–310.

[10] ———, *Simple randomized algorithms for closest pair problems*, Nordic J. Comput., 2 (1995), pp. 3–27.

[11] D. T. Lee, *Two-dimensional Voronoi diagrams in the L_p-metric*, J. Assoc. Comput. Mach., 27 (1980), pp. 604–618.

[12] G. S. Lueker, *A data structure for orthogonal range queries*, in Proc. 19th Annual IEEE Symposium on Foundations of Computer Science, IEEE Press, Piscataway, NJ, 1978, pp. 28–34.

[13] K. Mehlhorn, *Data Structures and Algorithms, Volume* 3: *Multi-Dimensional Searching and Computational Geometry*, Springer-Verlag, Berlin, 1984.

[14] K. Mehlhorn and S. Näher, *Dynamic fractional cascading*, Algorithmica, 5 (1990), pp. 215–241.

[15] F. P. Preparata and M. I. Shamos, *Computational Geometry: An Introduction*, Springer-Verlag, New York, 1985.

[16] J. S. Salowe, *Enumerating interdistances in space*, Internat. J. Comput. Geom. Appl., 2 (1992), pp. 49–59.

[17] C. Schwarz, *Data structures and algorithms for the dynamic closest pair problem*, Ph.D. thesis, Department of Computer Science, Universität des Saarlandes, Saarbrücken, Germany, 1993.

[18] C. Schwarz, M. Smid, and J. Snoeyink, *An optimal algorithm for the on-line closest pair problem*, Algorithmica, 12 (1994), pp. 18–29.

[19] M. I. Shamos and D. Hoey, *Closest-point problems*, in Proc. 16th Annual IEEE Symposium on Foundations of Computer Science, IEEE Press, Piscataway, NJ, 1975, pp. 151–162.

[20] M. Smid, *Maintaining the minimal distance of a point set in less than linear time*, Algorithms Review, 2 (1991), pp. 33–44.

[21] M. Smid, *Dynamic rectangular point location, with an application to the closest pair problem*, Inform. Comput., 116 (1995), pp. 1–9.

[22] ———, *Maintaining the minimal distance of a point set in polylogarithmic time*, Discrete Comput. Geom., 7 (1992), pp. 415–431.

[23] K. J. Supowit, *New techniques for some dynamic closest-point and farthest-point problems*, in Proc. 1st Annual ACM–SIAM Symposium on Discrete Algorithms, Society for Industrial and Applied Mathematics, Philadelphia, 1990, pp. 84–90.

[24] P. M. Vaidya, *An $O(n \log n)$ algorithm for the all-nearest-neighbors problem*, Discrete Comput. Geom., 4 (1989), pp. 101–115.

[25] D. E. Willard and G. S. Lueker, *Adding range restriction capability to dynamic data structures*, J. Assoc. Comput. Mach., 32 (1985), pp. 597–617.

[26] A. C. Yao, *On constructing minimum spanning trees in k-dimensional spaces and related problems*, SIAM J. Comput., 11 (1982), pp. 721–736.

SIAM J. COMPUT.
Vol. 25, No. 4, pp. 797–827, August 1996

005

EFFICIENT PARALLEL ALGORITHMS FOR CHORDAL GRAPHS*

PHILIP N. KLEIN[†]

Abstract. We give the first efficient parallel algorithms for recognizing chordal graphs, finding a maximum clique and a maximum independent set in a chordal graph, finding an optimal coloring of a chordal graph, finding a breadth-first search tree and a depth-first search tree of a chordal graph, recognizing interval graphs, and testing interval graphs for isomorphism. The key to our results is an efficient parallel algorithm for finding a perfect elimination ordering.

Key words. parallel algorithms, chordal graphs, interval graphs, PQ-tree, perfect elimination ordering

AMS subject classifications. 68Q20, 68R10, 68Q22, 68Q25

1. Introduction.

Chordal graphs are graphs in which every cycle of length > 3 has a *chord*, an edge between nonconsecutive nodes of the cycle. Chordal graphs have application in Gaussian elimination [39] and databases [4] and have been the object of much algorithmic study since the work of Fulkerson and Gross in 1965 [17].

Chordal graphs are an important subclass of the class of *perfect graphs* [5], [23], which are graphs in which the maximum clique size equals the chromatic number for every induced subgraph. No polynomial-time algorithm for recognizing perfect graphs is known. In contrast, chordal graphs can be recognized in linear time.

1.1. Our results.

In this paper, we give the first efficient parallel algorithms for a host of chordal-graph problems. Our deterministic algorithms take $O(\log^2 n)$ time and use only $n+m$ processors of a concurrent-read concurrent-write parallel random-access machine (CRCW PRAM) for n-node, m-edge graphs. Moreover, using randomized techniques [19], [33], [38], we can achieve the same time bound with only $(n+m)/\log n$ processors. Thus our algorithms are nearly optimal in their use of parallelism, in contrast to the previous parallel algorithms that required about n^3 processors to achieve the same time bound. The chordal graph problems we solve are as follows:

1. recognizing chordal graphs,
2. finding all maximal cliques in a chordal graph (and, in particular, finding a maximum-weight clique),
3. finding a maximum independent set (and a minimum clique cover) in a chordal graph,
4. finding an optimal coloring of a chordal graph,
5. finding a depth-first search tree of a chordal graph,
6. finding a breadth-first search tree of a chordal graph.

Chordal graphs include as a subclass *interval graphs*, the intersection graphs of intervals of the real line. Thus our algorithms for problems 2–6 above may be applied to interval graphs. But we can also solve two additional interval-graph problems. Namely, in $O(\log^2 n)$ time using $n + m$ processors, we can

- recognize interval graphs and find interval representations and
- test isomorphism between interval graphs.

*Received by the editors June 12, 1989; accepted for publication (in revised form) December 6, 1994. A preliminary version of this paper appeared in *Proc.* 1988 *Symposium on the Foundations of Computer Science*, pp. 150–161 [28], and in technical report TR-426, Laboratory for Computer Science, Massachusetts Institute of Technology, 1988. The research described in this paper is part of the author's Ph.D. thesis at MIT. This research was supported by a fellowship from the Center for Intelligent Control Systems, with additional support from Air Force Contract AFOSR-86-0078, a PYI awarded to David Shmoys with matching support from IBM and Sun Microsystems, and ONR grant N00014-08-K-0243 at Harvard University.

[†]Department of Computer Science, Brown University, Box 1910, Providence, RI 02912.

The isomorphism algorithm requires the CRCW PRAM to be of type "priority" (higher-numbered processors win in case of write conflicts). It makes use of an efficient parallel algorithm for tree isomorphism.

1.2. Other parallel algorithms. In a previous work, Naor, Naor, and Schäffer [34] gave parallel algorithms for chordal-graph problems 1–4. Their algorithm for problem 1 used $O(n^2m)$ processors. They also gave an algorithm for problem 2 that required $O(n^{5+\epsilon})$ processors to achieve $O(\log^2 n)$ time and $O(n^4)$ processors to achieve $O(\log^3 n)$ time. They showed how, subsequent to the solution of problem 2, problems 3 and 4 could be solved in $O(\log^2 n)$ additional time using $O(n^2)$ processors. Thus they identified problem 2 as a bottleneck in analyzing chordal graphs. Subsequent research (independent of and concurrent with our work) by Dahlhaus and Karpinski [12], [13] and Ho and Lee [24] reduced the processor bound for problem 2 to $O(n^4)$ and $O(n^3)$, respectively; because of the algorithms of [34], these processor bounds then apply also to problems 3 and 4. The algorithm of Ho and Lee required only $O(\log n)$ time.

A parallel algorithm for finding a depth-first search tree in an arbitrary graph was given by Aggarwal and Anderson [2]. Their algorithm is randomized and uses $O(nM(n))$ processors.[1] A parallel algorithm for breadth-first search in an arbitrary graph that uses $M(n)$ processors was given by Gazit and Miller [20] that uses $M(n)$ processors. Depth-first and breadth-first algorithms specifically for chordal graphs have not previously appeared in the literature.

To our knowledge, no previous NC algorithm was known for interval-graph isomorphism. Recognition of interval graphs was previously shown to be in NC by Kozen, Vazirani, and Vazirani [30], but no specific time or processor bound was given. Novick [37] has given an $O(\log n)$-time, n^3-processor CRCW algorithm for recognizing interval graphs. He has also claimed [36] an algorithm with the same bounds for constructing a PQ-tree representing a given interval graph. He observed [35] that this latter task is the first step in Lueker and Booth's interval-graph isomorphism algorithm and suggested that the remaining steps might be parallelizable. Savage and Wloka [41] have given an efficient parallel algorithm for optimum coloring of interval graphs. Their algorithm takes $O(\log n)$ time using n processors of an exclusive-read exclusive-write (EREW) PRAM, assuming that the interval representation of the graph has been provided.

1.3. Background. The key to our algorithmic results is our use of the *perfect elimination ordering* (PEO) of a graph, a node ordering that exists if and only if the graph is chordal. Fulkerson and Gross [17] discovered the PEO and used it to find all the maximal cliques of a chordal graph. Rose [39] has related the PEO to the process of Gaussian elimination in a sparse symmetric positive definite linear system. Rose, Tarjan, and Lueker [40] gave a linear-time algorithm for finding a PEO in a chordal graph using the notion of *lexicographic breadth-first search.* This yields a linear-time sequential algorithm for recognizing chordal graphs (problem 1). Once a PEO for a graph is known, algorithms due to Gavril [18] for problems 2–4 can be implemented in linear time. Thus the PEO has emerged as the key technique in sequential algorithms for chordal graphs, and its study has yielded important algorithmic ideas in the sequential realm.

Researchers in parallel algorithms, however, have largely abandoned use of the PEO—largely because finding a PEO in parallel seemed so difficult. The existence of an NC algorithm for finding a PEO algorithm was left open by Edenbrandt [15], [16] and by Chandrasekharan

[1] Here $M(n)$ denotes the time required to multiply two $n \times n$ matrices. The best bound known, due to Coppersmith and Winograd [11], is $O(n^{2.376})$.

and Iyengar [7] and resolved by Naor, Naor, and Schäffer [34] and independently by Dahlhaus and Karpinski [12], [13]. However, the PEO algorithms of [34] and [12] required at least n^4 processors (subsequently improved to n^3 by Ho and Lee [24], [25]). In fact, it is suggested in [34] that for parallel algorithms the PEO may be less useful than the representation of a chordal graph as the intersection graph of subtrees of a tree. The results of this paper suggest otherwise.

We describe a new parallel algorithm for finding a PEO. Our algorithm takes $O(\log^2 n)$ time and uses only a linear number of processors of a CRCW PRAM. In fact, we can achieve the same time bound using only $O((n+m)\log n)$ processors of a randomized PRAM. Thus our PEO algorithm is nearly optimal. The algorithm relies on a new understanding of the combinatorial nature of PEOs. The algorithm in turn forms the basis for our other efficient parallel algorithms for chordal and interval graphs.

Our algorithm for finding a PEO actually solves the following problem: given a labeling of the nodes of a graph with numbers (a *numbering*), the algorithm finds a PEO consistent with the partial order defined by the numbering or determines that no such consistent PEO exists. It accomplishes this by iteratively refining the numbering until each number is assigned to only one node. Our methods ensure that only $O(\log n)$ refinements suffice. Once the numbering is one to one, it is easy to check whether it defines a PEO, as we observe in §4.2.

Most of our algorithms for solving optimization problems on a chordal graph rely on a tree derived from the PEO, the *elimination tree*. We show that breaking up the tree by removing a vertex corresponds to breaking up the graph by removing a clique. We use this observation to give a divide-and-conquer algorithm for optimum coloring. In order to find a maximum independent set and a minimum clique cover of the graph, we apply terminal-branch elimination, a technique of Naor, Naor, and Schäffer [34], to the elimination tree. We believe the elimination tree may also prove useful in other parallel chordal-graph algorithms.

The interval-graph algorithms rely on a parallel algorithm, MREDUCE, for manipulating the PQ-data structure of Booth and Lueker [6]. This algorithm is described in §3.

1.4. Graph notation. Let G be an undirected graph. We use $V(G)$ to denote the set of nodes of G. Let H be a subgraph of G or a set of nodes of G. We use $G[H]$ to denote the subgraph of G induced by the nodes of H. We use $G - H$ to denote the subgraph obtained from G by deleting the nodes of H. We use $|H|$ to denote the number of nodes in H. Unless otherwise stated, n and m denote the number of nodes and number of edges, respectively, in the graph G.

2. The PEO algorithm.

The most algorithmically useful characterization of chordal graphs, the PEO, was discovered by Fulkerson and Gross [17] in 1965. Dirac had proved in [14] that every chordal graph has a *simplicial* node, a node whose neighbors form a clique. Fulkerson and Gross observed that since every induced subgraph of a chordal graph is also chordal, deletion of a vertex and its incident edges results in a chordal graph. They proposed an "elimination" process: repeatedly find a simplicial node and delete it until all nodes have been deleted or no remaining node is simplicial. It follows from Dirac's theorem that the process deletes every node of a chordal graph; in fact, Fulkerson and Gross showed conversely that a graph is chordal if the process deletes every node. Thus the elimination process constitutes an algorithm for recognition of chordal graphs. The order in which nodes are deleted is called a PEO.

2.1. An overview of the PEO algorithm.

We define a PEO of a graph G to be a one-to-one numbering $v_1, \ldots, v_n$ of the nodes of G such that for each i ($i = 1, \ldots, n$), the higher-numbered neighbors of v_i form a clique. We also represent a PEO as a sequence of nodes $\sigma = v_1 \ldots v_n$.

THEOREM 2.1 (Fulkerson and Gross). *A graph G has a PEO if and only if G is chordal.*

In order to give a parallel algorithm for finding a PEO of a chordal graph G, we generalize the notion to numberings that are not one to one. Let \$ be a numbering of the nodes of G (a function mapping nodes to numbers). We use $G_\$$ to denote the graph G with each node v labeled by its number $\$(v)$. We shall use the metaphor of wealth in connection with numberings \$; for example, if $\$(v) > \$(w)$, we shall say v is "richer" than w. The *classes* of $G_\$$ are the subgraphs induced on sets of equal-numbered nodes. The *class-components* of $G_\$$ are the connected components of the classes of $G_\$$.

Let \$ and £ be two numberings of the nodes of G. We say \$ is *consistent with £* (and £ is a *refinement of* \$) if $\$(v) < \$(w)$ implies $£(v) < £(w)$ for all nodes v and w. We say £ is a refinement of $G_\$$ if we wish to emphasize the graph for which £ is a numbering. Note that each class-component of $G_£$ is a subgraph of some class-component of $G_\$$.

We call ¢ a *partial numbering* if ¢ assigns numbers to *some* of the nodes of G, and is undefined for others; a numbering is trivially a partial numbering. For the numbering \$ and the partial numbering ¢, the *refinement of* \$ *by* ¢ is defined to be the numbering \$¢ in which ¢ is used to break ties in \$. That is, $\$¢(v) < \$¢(w)$ if either

- $\$(v) < \$(w)$ or
- $\$(v) = \$(w)$, $¢(v)$ and $¢(w)$ are defined, and $¢(v) < ¢(w)$.

Typically, ¢ will be a numbering of some class-component C of $G_\$$ and hence only a partial numbering of $G_\$$. In this case, we speak of obtaining \$¢ from \$ as *stratifying* the class-component C of $G_\$$, or as *well-stratifying* C if, in addition, each class-component of $G_{\$¢}$ contains at most $\frac{4}{5}|C|$ nodes of C.

We want to know when a numbering \$ is consistent with some PEO. To this end, we introduce the notion of a *backward path* in $G_\$$: namely, a simple path whose endpoints are strictly richer than all its internal nodes.[2] We say a numbering \$ of G is *valid* if every backward path in $G_\$$ has adjacent endpoints. The following lemmas are immediate from the definitions.

LEMMA 2.2. *For a graph G, if a valid numbering \$ of G is also one to one, then \$ is a PEO of G.*

LEMMA 2.3. *Let \$ be a valid numbering of a graph G. For any class-component C of $G_\$$, the richer neighbors of C form a clique.*

We assume for the remainder of this section that G is a connected chordal graph. Our algorithm for finding a PEO in G, which appears in Figure 1, consists of a sequence of $O(\log n)$ stages. In each stage, the algorithm modifies the numbering \$ by well-stratifying every nonsingleton class-component C while preserving the validity of \$, using a procedure STRATIFY(G\$, C). In each stage, the size of the largest class-component goes down by a factor of $\frac{4}{5}$. Hence after at most $\log_{5/4} n$ stages, the current numbering is one to one and the algorithm terminates, outputting the current numbering, which is a PEO by Lemma 2.2.

We shall show in §2.2 that the procedure STRATIFY($G_\$$, C) can be implemented to run in $O(\log k)$ time using k processors, where k is the number of edges that have at least one endpoint in C. Consequently, executing step R4 of ITERATED REFINEMENT for all class-components in parallel requires $O(\log m)$ time using $O(m)$ processors. Since the number of stages is $O(\log n)$, the total time required by ITERATED REFINEMENT is $O(\log^2 n)$. Using the

[2] The notion is a generalization of one appearing in Lemma 4 of [40].

```
ITERATED REFINEMENT
R1  To initialize, let $\$$ be the trivial numbering assigning 0 to every node of $G$.
R2  While $\$$ is not one to one,
R3    For each nonsingleton class-component $C$ of $G_{\$}$ in parallel:
R4      call STRATIFY($G_{\$}$, $C$).
```

FIG. 1. *The* ITERATED REFINEMENT *algorithm for finding a PEO.*

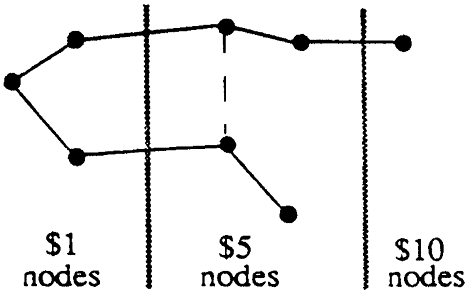


FIG. 2. *To obtain a uniform path, delete the nodes of backward subpaths.*

randomized connectivity algorithm of Gazit [19], we can reduce the processor bound by a factor of $\log n$ without increasing the time bound.

As an aside, we note that the initial numbering can be any valid numbering $\$$ of G, e.g., the trivial numbering assigning the same number to all nodes; the algorithm's output will then be a PEO consistent with $\$$. This observation leads to the following theorem, which is not needed for our algorithm, but which justifies our initial definition of validity.

BACKWARD-PATH THEOREM. *For a chordal graph G, a numbering $\$$ of G is valid if and only if it is consistent with some PEO of G.*

Proof. The "only if" direction will follow from the correctness of the algorithm. To prove the other direction, suppose σ is a PEO of G consistent with $\$$. We need to show that every backward path in $G_{\$}$ has adjacent endpoints. For the two endpoints x and y of any backward path, let P be the shortest backward path with these two endpoints. If P consists of the single edge $\{x, y\}$, we are done, so assume that P has internal nodes. Let u be the internal node with the minimum σ-number. Then the neighbors of u in P have higher σ-number, so they are adjacent by definition of a perfect ordering. Thus there is a shorter backward path connecting x and y, a contradiction. □

We return to the algorithm. The key to the efficiency of the procedure STRATIFY($G_{\$}$, C) is that it need only consider the graph induced by C and its richer neighbors in $G_{\$}$, as we shall show presently.

We say that a path in $G_{\$}$ is *weakly backward* if its endpoints are at least as rich as its internal nodes. When we wish to emphasize that a path P is backward, as opposed to only weakly backward, we shall say P is *strictly* backward. If $\pounds$ is a refinement of $\$$, a path P that is weakly backward in $G_{\pounds}$ is also weakly backward in $G_{\$}$. If P is strictly backward in $G_{\pounds}$ but not strictly backward in $G_{\$}$, then at least one of the endpoints of P has the same $\$$-number as one of the internal nodes of P. We say a path in G is *uniform* (with respect to a numbering $\$$) if every internal node has the same $\$$-number as the poorer of the two endpoints.

LEMMA 2.4. *Suppose $\$$ is a valid numbering of the graph G. For each weakly backward path P in $G_{\$}$, there is a uniform weakly backward path P' in $G_{\$}$ with the same endpoints such that $V(P') \subseteq V(P)$.*

Proof. The proof is illustrated in Figure 2. Let $v_1 \dots v_k$ be the nodes of a weakly backward path P, and let t be the $\$$-number of its poorer endpoint. Suppose $v_i \dots v_j$ is a maximal subpath consisting of nodes poorer than t. Then $v_{i-1} \dots v_{j+1}$ is a backward path in $G_\$$, so its endpoints v_{i-1} and v_{j+1} are adjacent by the validity of \$. We can therefore replace the subpath $v_{i-1} \dots v_{j+1}$ with the edge $\{v_{i-1}, v_{j+1}\}$, obtaining a backward path P_1 with the same endpoints as P but with fewer nodes poorer than t. Note that every node in P_1 is a node of P. Continuing this process yields a path with the same endpoints and with no nodes poorer than t and consisting of a subset of the nodes of P. □

The following lemma shows that STRATIFY($G_\$$, C) need only consider C and the higher-numbered neighbors of C. Fix the graph $G_\$$. For a class-component C, let $\widehat{C}$ denote the subgraph of $G_\$$ induced by C and its higher-numbered neighbors in $G_\$$.

REFINEMENT LEMMA. *Suppose \$ is a valid numbering of a graph G. Let ¢ be a numbering of a class-component C of $G_\$$. Then the refinement of $G_\$$ by ¢ is valid if and only if the refinement of $\widehat{C}_\$$ by ¢ is valid.*

Proof. Let \$¢ be the refinement of $G_\$$ by ¢, and let λ be the refinement of $\widehat{C}_\$$ by ¢. Assume that \$ is a valid numbering of G. If \$¢ is also a valid numbering of G, then λ is a valid numbering of $\widehat{C}$ because $\widehat{C}_\lambda$ is a node-induced subgraph of $G_{\$¢}$. Conversely, suppose that λ is a valid numbering of $\widehat{C}$. For each backward path P in $G_{\$¢}$, we must show that P's endpoints x and y are adjacent. Since P is weakly backward in $G_\$$, there exists a uniform path P' in $G_\$$ with endpoints x and y, where $V(P') \subseteq V(P)$. Since $V(P') \subseteq V(P)$, the path P' is still backward with respect to \$¢. If P' has no internal nodes, x and y are adjacent. Suppose P' has internal nodes; they all have the same \$-number as the lower-numbered endpoint x by definition of uniformity. Since P' is strictly backward with respect to \$¢, it follows that the internal nodes have a lower ¢-number than x. Hence the internal nodes and x must all lie in C because ¢ is defined only on C. Then the other endpoint y is a neighbor of a node of C and hence is either in C itself or is a higher-numbered neighbor of C. In the first case, y has a higher ¢-number than the internal nodes because P' is strictly backward. In the second case, y has a higher λ-number than the internal nodes. In either case, we conclude that P' is a backward path in $\widehat{C}_\lambda$. By the validity of λ, x and y are adjacent. □

The Refinement Lemma implies that to validly stratify a class-component C in $G_\$$, we need only validly stratify it in $\widehat{C}_\$$. In fact, we observe next that all the class-components of $G_\$$ may be thus stratified simultaneously and independently. To see this, let $C^1, \dots, C^k$ be the class-components of $G_\$$, ordered in some arbitrary way consistent with \$. We show that stratifying the class-components in order is equivalent to stratifying them all at once, as far as validity is concerned.

For $i = 1, \dots, k$, let $¢_i$ be a numbering of C^i such that the refinement of $\widehat{C^i}_\$$ by $¢_i$ yields a valid numbering of $\widehat{C^i}$. Let $\$_0 = \$$, and for $i = 1, \dots, k$, let $\$_i$ be the refinement of $\$_{i-1}$ by $¢_i$. The numbering that $\$_{i-1}$ induces on $\widehat{C^i}$ is isomorphic to that induced by \$ (the same order relations hold), so refining $\widehat{C^i}_{\$_{i-1}}$ by $¢_i$ is valid. It then follows via the Refinement Lemma that $\$_i$ is valid. We conclude that $\$_k$ is a valid refinement of \$. We can obtain $\$_k$ directly by using $¢_i$ to stratify C^i for all class-components C^i in parallel. This is how steps R3 and R4 of ITERATED REFINEMENT are carried out. The algorithm STRATIFY($G_\$$, C^i) for stratifying C^i is given in §2.2.

2.2. Valid well-stratification. Before we give the algorithm for valid well-stratification, we give some results used in proving the correctness of the algorithm. We start with a lemma of Dirac [14].

LEMMA 2.5 (Dirac). *If S is a minimal set of nodes whose removal separates a connected chordal graph into exactly two connected components, then S is a clique.*

COROLLARY 2.6. *In a chordal graph, the common neighbors of two nonadjacent nodes form a (possibly empty) clique.*

Proof. In the induced subgraph consisting of the two nonadjacent nodes x and y and their common neighbors, the common neighbors form a minimal separator between x and y. □

Corollary 2.7. *Let H be a connected subgraph of a chordal graph G. Then the numbering δ is valid, where δ assigns* 1 *to H and neighbors of H and* 0 *to all other nodes.*

Proof. Let P be a backward path in G_δ, and let C be the component of the 0-numbered nodes that contains the internal nodes of P. Let D be the set of 1-numbered neighbors of nodes in C. In the subgraph induced on $C \cup D \cup H$, the nodes of D form a minimal separator between C and H, so D is a clique. The endpoints of P are in D, so they are adjacent. □

Lemma 2.8. *Let K be a clique in a chordal graph G. Then the numbering δ is valid, where δ assigns* 1 *to K and those nodes adjacent to all of K and* 0 *to all other nodes.*

Proof. This is a proof by induction on $|K|$. The base case, in which $|K| = 1$, follows from Corollary 2.7. Suppose $|K| > 1$, and let v be a node of K. Let A consist of the nodes of $K - \{v\}$ and the nodes adjacent to all of $K - \{v\}$. Let α be the numbering assigning 1 to A and 0 to other nodes. By the inductive hypothesis, α is valid. Because K is a clique, v is in A. Let β be the numbering of A that assigns 2 to v and its neighbors and 1 to other nodes of A. By Corollary 2.7, β is a valid numbering of A. Let γ be the refinement of α by β. The nodes of A have no richer neighbors in G_α, so by the Refinement Lemma, γ is a valid numbering of G. But γ is a refinement of the numbering δ defined in the statement of the lemma, so δ is also valid. This completes the inductive step. □

Lemma 2.9. *Let α be any valid numbering of a graph G, and let K be a clique contained in the highest-numbered class of G_α. Suppose the numbering γ is obtained from α by increasing the numbers of nodes of K. Then γ is valid.*

Proof. The only backward paths introduced have endpoints in the clique K. □

Lemma 2.10. *Let* \$ *be a valid numbering of a graph G. Suppose C is a class-component of $G_\$$, and all nodes in C have the same richer neighbors in $G_\$$. Let ¢ be a valid numbering of C. Then the refinement of $G_\$$ by ¢ is valid.*

Proof. By the Refinement Lemma, we need only show that the refinement λ of $\widehat{C}_\$$ by ¢ preserves validity. Assume \$ is valid, so the nodes of $\widehat{C}$ not in C form a clique by Lemma 2.3. Let P be a backward path in $\widehat{C}_\lambda$ with endpoints x and y. We must show that x and y are adjacent. If both endpoints are in C, they are already adjacent by the validity of ¢. If neither is in C, they belong to a clique and hence are adjacent. Suppose therefore that x is in C and y is not. Since P is a backward path and an endpoint lies in C, all the internal nodes of P must also lie in C. Hence y is a richer neighbor of C in $\widehat{C}_\$$. Since all nodes in C have the same richer neighbors, it follows that y is a neighbor of x. □

The procedure Stratify $(G_\$, C)$ appears in Figure 3. If C is a nonsingleton class-component of $G_\$$, the procedure increases the numbers of some of the nodes of C, resulting in a valid refinement of $G_\$$ in which C has been well-stratified. The procedure takes $O(\log k)$ time using $O(k)$ processors, where k is the number of edges of G with at least one endpoint in C. To achieve this processor bound, the procedure first identifies these edges by inspecting the adjacency lists of nodes in C and subsequently never examines any other edges of G. While inspecting the adjacency lists, the procedure also identifies the set B of richer neighbors of C in $G_\$$. The procedure uses the fact that if \$ is a valid numbering of G, then the set of nodes B form a clique by Lemma 2.3. The procedure assumes the existence of edges between nodes in B without ever checking for their presence. Specifically, in computing connected components of a graph involving nodes of B, the procedure uses the algorithm of Shiloach and Vishkin [43], suitably modified to take into account the fact that any two nodes of B are adjacent.

Depending on the nodes in B, the procedure Stratify$(G_\$, C)$ calls one of three subprocedures, in which most of the work is done. In each procedure, we make use of *parallel prefix*

```
Procedure STRATIFY($G_\$$, $C$).
S1   For each node $v$ in $C$, identify those edges connecting $v$ to another node in $C$, and those
     edges connecting $v$ to a richer node.
S2   Let $B$ be the set of richer neighbors of $C$.
S3   If $B$ is empty, call NONE($G_\$$, $C$, 1), and end.
S4   Let $\epsilon$ be $1/2|C|$ times the difference between $\$(C)$ and the number assigned to the next
     higher class.
S5   If every node in $B$ has at least $\frac{2}{5}|C|$ neighbors in $C$, call HIGHDEGREE($G_\$$, $C$, $B$, $\epsilon$), and
     end.
S6   If there are nodes in $B$ with fewer than $\frac{2}{5}|C|$ neighbors in $C$, call LOWDEGREE($G_\$$, $C$, $B$, $\epsilon$).
```

FIG. 3. *The main procedure for finding a valid well-stratification.*

computation, due to Ladner and Fischer [31]. In particular, the subprocedures increase the numbers assigned to some of the nodes of C. We must make certain that these numbers are not increased too much. The new numbers to be assigned to nodes of C must all be less than numbers currently assigned to nodes richer than C; otherwise, the resulting numbering would not be a refinement of the old numbering. Therefore, in each procedure, we let ϵ denote a number small enough that the numbers of nodes of G richer than C exceed the numbers of nodes of C by at least $|C|\epsilon$. This choice of ϵ allows us to increase the numbers of C by up to $|C|\epsilon$ and still end up with a refinement of the old numbering. We find this convention useful in presenting the algorithms; however, see Implementation Note 1.

We now show that STRATIFY($G_\$$, C) succeeds in finding a valid refinement that well-stratifies C. We shall refer to the nodes of C as *crimson* nodes and the nodes of B as *blue* nodes. We consider three cases, corresponding to the three procedures.

Case I. There are no blue nodes. In this case, procedure NONE($G_\$$, C, ϵ) shown in Figure 4 is called. The procedure first identifies the set D of high-degree nodes: $D = \{v \in C : v \text{ has} > \frac{3}{5}|C| \text{ neighbors in } C\}$. The procedure then branches according to the following subcases.

Subcase (1). Some component H of $C - D$ has size $> \frac{4}{5}|C|$. In this case, the procedure uses the spanning tree T of H to choose a connected subgraph H' of H such that the set A consisting of H' and its neighbors includes between $n/5$ and $4n/5$ nodes. Then the numbers assigned to nodes of A are increased, resulting in a well-stratification of C. The validity of the numbering follows from Lemma 2.7.

Subcase (2). Each component of $C - D$ has size at most $\frac{4}{5}|C|$, and D is a clique. In this case, we increase the numbers assigned to nodes of D, placing each node of D in its own class. The components of the remaining nodes of C are small, so we have well-stratified C. The validity of the numbering follows from Lemma 2.9.

Subcase (3). D is not a clique. In this case, we choose two nonadjacent nodes x and y in D. At most $\frac{2}{5}|C|$ nodes of C are not neighbors of x, and so at least $\frac{1}{5}|C|$ neighbors of y are also neighbors of x. We increase the numbers of these common neighbors of x and y, putting each in its own class. The numbering is valid by Lemmas 2.6 and 2.9.

Case II. Each blue node is adjacent to at least $\frac{2}{5}|C|$ crimson nodes. In this case, the procedure HIGHDEGREE($G_\$$, C, B, ϵ) of Figure 5 is called. The procedure arbitrarily orders the blue nodes: $B = \{v_1, \ldots, v_k\}$. For $1 \leq j \leq k$, let F_j be the set of nodes v such that v is adjacent to all the nodes $v_1, \ldots, v_j$. Then $\hat{j}$ is chosen to be the maximum j such that F_j contains at least $|C|/5$ crimson nodes. The numbers of nodes in $F_{\hat{j}}$ are then increased by ϵ. The validity of the resulting numbering follows from Lemmas 2.8 and 2.9. At most $\frac{4}{5}|C|$ nodes of C do not have their numbers increased. However, the set $F_{\hat{j}}$ may be quite large. We consider two subcases.

```
Procedure NONE($G_\$$, $C$, $\epsilon$).
N1   Let $D = \{v \in C : v$ has $> \frac{3}{5}|C|$ neighbors in $C\}$.
N2   Find a spanning forest of $C - D$.
N3   If some component $H$ of $C - D$ has at least $\frac{4}{5}|C|$
     nodes,
N4     Let $T$ be the spanning tree of $H$.
N5     Arrange the nodes of $H$ in some order consistent with their distance in $T$ from the root:
       $v_1, \ldots, v_k$.
N6     For $1 \leq j \leq k$, let $A_j$ denote the set consisting of $v_1, \ldots, v_j$ and neighbors of these
       nodes in $C$.
N7     Using parallel prefix computation,
       choose $\hat{j} = \max\{j : |A_j| \leq \frac{4}{5}|C|\}$.
N8     Increase the numbers of nodes in $A_{\hat{j}}$ by $\epsilon$.
N9   Otherwise,
N10    If $D$ is a clique,
N11      Let $v_1, \ldots, v_k$ be the nodes of $D$.
N12      For $1 \leq i \leq k$, add $\epsilon i$ to $v_i$'s number.
N13    Otherwise,
N14      Let $x$ and $y$ be two nonadjacent nodes of $D$.
N15      Let $v_1, \ldots, v_k$ be their common neighbors.
N16      For $1 \leq i \leq k$, add $\epsilon i$ to $v_i$'s number.
```

FIG. 4. *The subprocedure used to well-stratify C if C has no richer neighbors.*

```
Procedure HIGHDEGREE($G_\$$, $C$, $B$, $\epsilon$).
(Each node of $B$ has at least $\frac{2}{5}|C|$ neighbors in $C$.)
H1   Arbitrarily order the nodes of $B$: $v_1, \ldots, v_k$.
H2   for $1 \leq j \leq k$, let $F_j$ denote the set of nodes of $C$
     adjacent to all of the nodes $v_1, \ldots, v_j$.
H3   Using parallel prefix computation,
     choose $\hat{j} = \max\{j : |F_j| \geq \frac{1}{5}|C|\}$.
H4   Increase the numbers of nodes in $F_{\hat{j}}$ by $\epsilon$.
H5   Let $C'$ be the largest component of $G[F_{\hat{j}}]$.
H6   If $\hat{j} = k$, then call NONE($G_\$$, $C'$, $\epsilon$).
```

FIG. 5. *The procedure* HIGHDEGREE *used to stratify C in case every richer neighbor of C has high degree in C.*

Subcase (1). $\hat{j} < k$. In this case, by choice of $\hat{j}$, the set of crimson nodes adjacent to all the nodes $v_1, \ldots, v_{\hat{j}+1}$ is less than $|C|/5$. Since there are at most $\frac{3}{5}|C|$ crimson nodes not adjacent to $v_{\hat{j}+1}$, it follows that the number of nodes adjacent to all the nodes $v_1, \ldots, v_{\hat{j}}$ is less than $\frac{1}{5}|C| + \frac{3}{5}|C| = \frac{4}{5}|C|$. Thus C has been well-stratified in this case.

Subcase (2). $\hat{j} = k$. In this case, every node in $F_{\hat{j}}$ is adjacent to every blue node. The procedure finds the largest component C' of $G[F_{\hat{j}}]$ and calls NONE($G_\$$, C', ϵ), which stratifies C' as if C' had *no* richer neighbors. The validity of the resulting numbering of $\widehat{C}$ follows from Lemma 2.10 and the Refinement Lemma. Since C' is well-stratified and $|C - C'| \leq \frac{4}{5}|C|$, C is well-stratified.

Case III. Neither Case I nor Case II holds. In this case, the procedure LOWDEGREE($G_\$$, C, B, ϵ) in Figure 6 is called. The procedure defines D to be the set of nodes in $C \cup B$ having more than $\frac{3}{5}|C|$ crimson neighbors. The procedure then finds a spanning tree T of the (unique) component H of $G[(C \cup B) - D]$ containing blue nodes and roots T at a blue node. The nodes of T are arranged in some order consistent with their distance from the root: $v_1, \ldots, v_k$. For $1 \leq j \leq k$, let A_j be the set consisting of $v_1, \ldots, v_j$ and neighbors of

```
Procedure LOWDEGREE($G_{\$}$, $C$, $B$, $\epsilon$).
(There exists a node in $B$ having fewer than $\frac{2}{5}|C|$ neighbors in $C$.)
L1   Let $D = \{v \in C \cup B : v$ has $> \frac{3}{5}|C|$ neighbors in $C\}$.
L2   Let $H$ be the connected component of $G[C \cup B - D]$
     containing nodes of $B$.
L3   Find a spanning tree $T$ of $H$, rooted at a node
     of $B$.
L4   Arrange the nodes of $T$ in some order consistent with their distance in $T$ from the root:
     $v_1, \ldots, v_k$.
L5   For $1 \leq j \leq k$, let $A_j$ denote the set consisting of
     $v_1, \ldots, v_j$ and neighbors of these nodes in $C$.
L6   Using parallel prefix computation,
     choose $\hat{j} = \max\{j : |A_j \cap C| \leq \frac{4}{5}|C|\}$.
L7   Increase the numbers of nodes in $A_{\hat{j}} \cap C$ by $\epsilon$.
L8   Let $C'$ be the largest component of $C - A_{\hat{j}}$.
L9   If $|C'| > \frac{4}{5}|C|$, then call STRATIFY($G_{\$}$, $C'$).
```

FIG. 6. *The procedure* LOWDEGREE *used to stratify C in case some richer neighbor of C has low degree in C.*

these nodes in $\widehat{C}$. Then $\hat{j}$ is chosen to be the maximum j such that A_j includes at most $\frac{4}{5}|C|$ crimson nodes.

Next, the numbers of nodes in $A_{\hat{j}}$ are increased in step L7. To see that the resulting numbering is valid, first consider the intermediate numbering in which all nodes in $A_{\hat{j}}$ have the same number, a number higher than that assigned to the nodes in $\widehat{C}-A_{\hat{j}}$. Since $H[\{v_1, \ldots, v_{\hat{j}}\}]$ is connected, the intermediate numbering is valid by Lemma 2.7. To obtain the numbering produced in step L7 from the intermediate numbering, we need only increase the numbers of some blue nodes. The blue nodes form a clique lying in $A_{\hat{j}}$, so the validity of the final numbering follows from Lemma 2.9.

The set $A_{\hat{j}} \cap C$ of nodes whose numbers have increased has size at most $\frac{4}{5}|C|$. However, the set $C - A_{\hat{j}}$ of nodes whose numbers have not increased may be quite large. The procedure finds the largest component C' of $C - A_{\hat{j}}$ and proceeds according to the following two subcases.

Subcase (1). $|C'| \leq \frac{4}{5}|C|$. In this case, we are done; C has been well-stratified.

Subcase (2). $|C'| > \frac{4}{5}|C|$. First, we observe that in this case, $\hat{j} = k$. To see this, suppose $\hat{j} < k$. Then the number of crimson nodes among $\{v_1, \ldots, v_{\hat{j}+1}\}$ and neighbors is more than $\frac{4}{5}|C|$. Since $v_{\hat{j}+1}$ is adjacent to at most $\frac{3}{5}|C|$ crimson nodes, it follows that the number of nodes whose numbers have increased is more than $\frac{4}{5}|C| - \frac{3}{5}|C| = \frac{1}{5}|C|$, which contradicts the fact that $|C'| > \frac{4}{5}|C|$.

Since $\hat{j} = k$, every crimson node in T and every crimson neighbor of T has had its number increased. Therefore, C' contains no nodes of T and no neighbors of T. Every node in D has more than $\frac{3}{5}|C|$ crimson neighbors, and all but at most $\frac{1}{5}|C|$ of the crimson nodes are in C', so every node in D has more than $\frac{2}{5}|C|$ crimson neighbors in C'. Thus every richer neighbor of C' is adjacent to at least $\frac{2}{5}|C|$ nodes of C'. This shows that the recursive call to STRATIFY in step L9 results in a call to HIGHDEGREE and not in a call to LOWDEGREE. Thus no further recursive calls occur. The recursive call well-stratifies C' and hence C as well, since $|C - C'| \leq \frac{1}{5}|C|$. The validity of the resulting numbering follows from the Refinement Lemma.

Implementation Note 1. We have described the procedures ITERATED REFINEMENT and STRATIFY as if real numbers were used to number the nodes. In implementing the algorithm, however, it is desirable to use integers to number the nodes. The simple approach is to multiply all numbers by n at the beginning of each stage and then renumber by sorting at the end of each stage. A more efficient approach is to intially allocate $4\log_{5/4} n$ bits for each node label, four bits per stage of ITERATED REFINEMENT. For the ith stage, we use the $4(i-1)+1$st through the $4i$th most significant bits. In Case I, Subcases (1) and (2), the procedure needs more bits; the procedure must assign a different number to each node of a clique. For each of these nodes, however, we can afford to use all the remaining bits because each such node ends up in its own class and hence needs no further labeling in subsequent stages.

Implementation Note 2. In step N7 of procedure NONE, we ordered the nodes $v_1, \ldots, v_k$ of a spanning tree T and chose $\hat{\jmath}$ maximum such that $\{v_1, \ldots, v_{\hat{\jmath}}\}$ and neighbors comprise at most $\frac{4}{5}|C|$ nodes. Here we provide more details for implementing that step; the techniques are also applicable to step H3 of procedure HIGHDEGREE and to step L6 of procedure LOWDEGREE. For each node $v \neq v_1$ in $\widehat{C}$, let *earliest-nbr*(v) be the minimum i such that v is adjacent to v_i or *undefined* if v has no neighbors among $v_1, \ldots, v_k$. Let *earliest-nbr*$(v_1) = 1$. For each node v_i in T, *earliest-nbr*$^{-1}(v_i)$ is the set of neighbors of v_i in $\widehat{C}$ that are not neighbors of any lower-numbered node. Let $f(v_i) = |$*earliest-nbr*$^{-1}(v_i)|$, and let $g(\ell) = \sum_{i=1}^{\ell} f(v_i)$ for $\ell = 1, \ldots, k$. Then $g(\ell)$ is the number of nodes comprised by $v_1, \ldots, v_\ell$ and their neighbors in $\widehat{C}$. The function $g(\cdot)$ can be computed from $f(\cdot)$ by a parallel prefix computation, after which $\hat{\jmath}$ is chosen as large as possible such that $g(\hat{\jmath}) \leq \frac{4}{5}|C|$.

Implementation Note 3. In step L2 of procedure LOWDEGREE, we computed the connected components of an induced subgraph of G containing nodes of C and nodes of B. We want to implement this step in such a way that the actual edges between nodes of B are not involved. To carry out the connected-components computation (and to find spanning trees of the components), we use the connectivity algorithm of [43], suitably modified to take into account our assumption that every two blue nodes are adjacent: we start by constructing a tree containing all the blue nodes in $H - D$ and another tree containing all the blue nodes in D. We then execute the algorithm of [43], using these artificial edges of these trees as surrogates for the set of edges between nodes in B. Thus the number of processors required is no more than the sum of $|C \cup B|$, the number of edges in C, and an additional term of $|B| - 1$ for the artificial edges.

This completes the description of the algorithm STRATIFY for valid well-stratification of a class-component. At most one recursive call is made, as we have shown. The time for the algorithm is dominated by the time to compute connected components and find spanning trees, which is $O(\log |C \cup B|)$ using the algorithm of [43]; as shown in Implementation Note 3, we need only $|C \cup B| + |E(C)| + |B| - 1$ processors, a number of processors bounded by at most twice the number of edges with at least one endpoint in C. We thus obtain the following theorem.

THEOREM 2.11. *Suppose* \$ *is a valid numbering of a chordal graph* G *and* C *is a class-component of* $G_\$$. *Valid well-stratification of* C *can be done in* $O(\log k)$ *time using* $O(k)$ *processors of a CRCW PRAM, where* k *is the number of edges with at least one endpoint in* C. *Hence, a PEO of the chordal graph* G *can be found in* $O(\log^2 n)$ *time using* $O(m)$ *processors.*

Using our algorithm for valid well-stratification in the procedure ITERATED REFINEMENT, we can therefore find a PEO of a graph G in $O(\log^2 n)$ time using $O(n+m)$ processors.

3. PQ-trees. In this section, we review the PQ-tree data structure developed by Booth and Lueker [6]. This data structure is useful in recognition and isomorphism-testing of interval graphs, problems we address in §4. In §3.1, we introduce a parallel PQ-tree-processing algorithm that arises in parallel algorithms for interval graph recognition and isomorphism-testing (§4).

A PQ-tree is a data structure developed by Booth and Lueker [6] for representing large sets of orderings of a ground set S. A PQ-tree T over the ground set S is a rooted tree whose leaves are the elements of S; every internal node is designated either a P-node or a Q-node and has at least two children. Hence T has at most $2n - 1$ nodes. The children of each internal node are ordered from left to right. These orderings induce a left-to-right ordering on the leaves of the tree; the sequence of leaves is called the *frontier* of the tree T and is denoted $\mathrm{fr}(T)$.

Let us say an automorphism of a PQ-tree T is *legal* if for every internal node v,

- if v is a Q-node then the automorphism either reverses the order of v's children or leaves the order unchanged, and
- if v is a P-node then the automorphism arbitrarily permutes the order of v's children.

The set of orderings of the ground set represented by a PQ-tree T is defined as

$$L(T) = \{\mathrm{fr}(T') \ : \ T' \text{ is obtained from } T \text{ by a legal automorphism}\}.$$

Consider, for example, a PQ-tree tree consisting of a P-node root whose children are all the leaves. This PQ-tree represents the set of all orderings of the ground set S and is therefore called the *universal* PQ-tree for S. Note that for any PQ-tree T, the automorphism that reverses the order of children of every node is legal, so if λ is in $L(T)$ then the reverse of λ is also in $L(T)$.

A special *null* PQ-tree is defined to represent the empty set of orderings.

Let A be a subset of S. An ordering λ of the elements of S is said to *satisfy* A if the elements of A form a consecutive subsequence of λ. For the PQ-tree T, let $\Psi(T, A) = \{\lambda \in L(T) : \lambda \text{ satisfies } A\}$.

Booth and Lueker give an algorithm REDUCE(T, A) that transforms T into a PQ-tree T' such that $L(T') = \Psi(T, A)$. We call this *reducing* T with respect to the set A. In this context, we call A a *reduction set.* Note that reduction can yield the null tree. The algorithm of Booth and Lueker takes $O(|A|)$ time. Each PQ-tree for a ground set S can be obtained from the universl PQ-tree by a series of reductions. Moreover, given any nonnull PQ-tree, it is easy to read off one of the orderings represented, namely the frontier of the tree. Klein and Reif [29] gave an algorithm MDREDUCE$(T, \{A_1, \dots, A_k\})$ that reduces T with respect to all the nonempty sets A_i simultaneously in the case where the sets A_i are all pairwise disjoint. The algorithm MDREDUCE runs in $O(\log n)$ time using n processors, where n is the size of the ground set of T. The case where the reduction sets are not disjoint was handled in [29], but the algorithm required $O(kn)$ processors and $O(\log k \log^2 n)$ time

In §3.1, we give a parallel reduction algorithm that handles nondisjoint reduction sets and requires only a linear number of processors. More specifically, the algorithm MREDUCE reduces a PQ-tree T with respect to nonempty subsets $A_1, \dots, A_k$ in time $O(\log n \cdot (\log n + \log m))$ time using $n + m$ processors, where $m = \sum_i A_i$. A preliminary version of this algorithm was given in [27].

3.1. PQ-tree nondisjoint reduction. The reduction algorithm uses a divide-and-conquer strategy in which recursive calls are made to different parts of the tree in parallel. At each level of recursion, a constant number of calls are made to a subroutine for reduction with respect to disjoint sets. These calls serve two purposes. The purpose of one call is to separate out parts of the tree from each other so that the algorithm can recur on them in parallel. Some of the the reduction sets A_i are relevant to two parts of the tree and thus to two recursive calls; such a set gives rise to two subsets, one for each call. Dividing A_i into two subsets and reducing the parts of the tree with respect to these subsets, however, does not completely solve the problem. One must introduce some additional constraints, effectively "gluing" the subsets together insofar as they constrain the PQ-tree. Thus the purpose of two other calls to the subroutine for disjoint reduction is to reduce the tree with respect to two special "gluing" sets that are derived from the original reduction sets A_i.

We first give Lemmas 3.1 and 3.2, which describe some simple properties of orderings. We next give a procedure GLUE which derives two sets from reduction sets A_i. The key property of these gluing sets is described in Lemma 3.3. Next, in Lemma 3.4, we show that reducing with respect to the sets A_i is equivalent to reducing with respect to subsets of these sets that lie in different parts of the tree (and also reducing with respect to the gluing sets). A key subroutine, SUBREDUCE, is then presented that is based on Lemma 3.4. The main algorithm of this section, MREDUCE, is mutually recursive with SUBREDUCE.

LEMMA 3.1. *Suppose λ satisfies A and B. Then*
intersection property: *λ satisfies $A \cap B$;*
union property: *if $A \cap B \neq \emptyset$, then λ satisfies $A \cup B$;*
difference property: *if $A \not\supseteq B$, then λ satisfies $A - B$.*

LEMMA 3.2. *Suppose λ satisfies sets B and C, and the leftmost symbol of B in λ coincides with the leftmost symbol of C in λ. Then either $B \subseteq C$ or $C \subseteq B$.*

Proof. Write $\lambda = \ldots \epsilon_1 \epsilon_2 \ldots$, where ϵ_1 is the leftmost symbol of B in λ and the leftmost symbol of C in λ. Then the subsequence of elements of B in λ is $\epsilon_1 \ldots \epsilon_{|B|}$ and the subsequence of elements of C is $\epsilon_1 \ldots \epsilon_{|C|}$. Thus if $|B| \leq |C|$ then $B \subseteq C$ and if $|C| \leq |B|$ then $C \subseteq B$. □

We say two sets A and B have a *nontrivial intersection* if the sets $A \cap B$, $A - B$, and $B - A$ are all nonempty.

The procedure to construct the gluing sets is as follows.

```
GLUE($E, \{A_1, \ldots, A_k\}$).
  • Let $\mathcal{A}$ be the collection of sets in $\{A_1, \ldots, A_k\}$ that have a nontrivial intersection
    with $E$.
  • If $\mathcal{A}$ is empty, return the pair $(\emptyset, \emptyset)$.
  • Let $A_p$ be a set in $\mathcal{A}$ that minimizes $|A_p \cap E|$.
  • Let $A_q$ be a set in $\mathcal{A}$ that minimizes $|A_q - E|$. subject to the constraint $A_q \cap E \supseteq$
    $A_p \cap E$.
  • Let $D = A_p \cap A_q$.
  • Let $\mathcal{A}'$ be the collection of sets $A_i$ in $\mathcal{A}$ such that $A_i \cap E \not\supseteq A_p \cap E$.
  • If $\mathcal{A}'$ is empty then return $(D, \emptyset)$.
  • Let $A_r$ be a set in $\mathcal{A}'$ that minimizes $|A_r \cap E|$.
  • Let $A_s$ be a set in $\mathcal{A}'$ that minimizes $|A_s - E|$ subject to the constraint $A_s \cap E \supseteq A_r \cap E$.
  • Let $F = A_r \cap A_s$.
  • Return $(D, F)$.
```

The sets are represented by a bipartite graph. Each ground-set element and each set A_i is a vertex. There is an edge between the ground-set element x and the set A_i if $x \in A_i$. To determine which sets A_i have a nontrivial intersection with E, we first mark the ground-set elements that belong to E. Then we determine, for each set A_i, how many marked elements A_i is adjacent to in the bipartite graph. If the number is bigger than zero but smaller than $|A_i|$ and smaller than $|E|$, then A_i has a nontrivial intersection with E. The other steps of the algorithm GLUE can be implemented in a similar way, using a constant number of marking and counting operations and finding the minimum among k numbers.

Let n denote the size of the ground set and let $m = \sum_{i=1}^{k} |A_i|$. Then the size of the bipartite graph is $O(n + m)$, and each such operation can be done in $O(\log(n + m))$ time using $(n + m)/\log(n + m)$ processors. Thus GLUE$(E, \{A_1, \ldots, A_k\})$ takes $O(\log(n + m))$ time using $(n + m)/\log(n + m)$ processors.

The proof of the following lemma is somewhat technical and can be skipped on a first reading.

LEMMA 3.3. *Suppose there exists some ordering satisfying $E, A_1, \ldots, A_k$. Let $(D, F) =$ $\text{GLUE}(E, \{A_1, \ldots, A_k\})$. For $i = 1, \ldots, k$, if A_i has a nontrivial intersection with E, then either $D \subseteq A_i$ or $F \subseteq A_i$.*

Proof. Let σ be the ordering satisfying $E, A_1, \ldots, A_k$. Let us write

$$\sigma = \ldots \alpha_1 \epsilon_1 \ldots \epsilon_2 \alpha_2 \ldots,$$

where α_1 is the last symbol before the elements of E, α_2 is the first symbol after the elements of E, and ϵ_1 and ϵ_2 are, respectively, the first and last symbols of E in σ.

Suppose $\mathcal{A}$ is nonempty. In this case, $D = A_p \cap A_q$. Since A_p contains at least one element of E and at least one symbol not in E, it contains two adjacent symbols, one in E and one not in E. Thus either $\alpha_1, \epsilon_1 \in A_p$ or $\alpha_2, \epsilon_2 \in A_p$. Assume without loss of generality that $\alpha_1, \epsilon_1 \in A_p$ (else replace σ with the reverse of σ). Since $A_q \cap E \supseteq A_p \cap E$, we have $\epsilon_1 \in A_q$. Since A_q also contains at least one symbol not in E, either $\alpha_1 \in A_q$ or $\alpha_2 \in A_q$. Since A_q is satisfied by σ, if α_2 were in A_q then all of E would also be in A_q, contradicting the fact that A_q has a nontrivial intersection with E. Thus $\alpha_1 \in A_q$. We conclude that if D is defined then $\alpha_1, \epsilon_1 \in D$.

Furthermore, since α_1 is the rightmost symbol in σ of both $A_p - E$ and $A_q - E$, by applying Lemma 3.2 to the reverse of σ, we infer that either $A_q - E \subseteq A_p - E$ or $A_p - E \subset A_q - E$. The second inclusion would imply $|A_p - E| < |A_q - E|$, which would contradict the choice of A_q. Thus we have $A_q - E \subseteq A_p - E$. By choice of A_q, we have $A_p \cap E \subseteq A_q \cap E$. We conclude that $(A_q - E) \cup (A_p \cap E) = A_q \cap A_p$, which in turn is D.

Suppose $\mathcal{A}'$ is also nonempty, so $F = A_r \cap A_s$. An analogous argument shows that either $\alpha_1, \epsilon_1 \in F$ or $\alpha_2, \epsilon_2 \in F$. Assume for a contradiction that the former holds. Then ϵ_1 is the leftmost symbol in σ of $A_p \cap E$ and of $A_r \cap E$. Hence by Lemma 3.2, either $A_p \cap E \subseteq A_r \cap E$ or $A_r \cap E \subset A_p \cap E$. The first inclusion would violate the choice of A_r. The second inclusion would imply $|A_r \cap E| < |A_p \cap E|$, which would violate the choice of A_p. This proves that if F is defined then $\alpha_2, \epsilon_2 \in F$.

To complete the proof, suppose A_i has a nontrivial intersection with E. We must show that either $D \subseteq A_i$ or $F \subseteq A_i$.

Since $\mathcal{A}$ is nonempty (it certainly contains A_i), $D = A_p \cap A_q$. Since σ satisfies A_i, which contains some symbols in E and some not in E, either $\alpha_1, \epsilon_1 \in A_i$ or $\alpha_2, \epsilon_2 \in A_i$.

Case I. $\alpha_1, \epsilon_1 \in A_i$. Then ϵ_1 is the first symbol in σ of both $A_i \cap E$ and $A_p \cap E$. By Lemma 3.2, therefore, either $A_p \cap E \subseteq A_i \cap E$ or $A_i \subset A_p \cap E$. The second inclusion would imply $|A_i \cap E| < |A_p \cap E|$, contradicting the choice of A_p. Hence we have $A_p \cap E \subseteq A_i \cap E$. Also, α_1 is the last symbol in σ of both $A_i - E$ and $A_q - E$. By applying Lemma 3.2 to the reverse of σ, we infer that either $A_q - E \subseteq A_i - E$ or $A_i - E \subset A_q - E$. The second inclusion would imply $|A_i - E| < |A_q - E|$, contradicting the choice of A_q. Hence we have $A_q - E \subseteq A_i - E$. We infer $(A_q - E) \cup (A_p \cap E) \subseteq (A_i - E) \cup (A_i \cap E)$. Thus $D \subseteq A_i$.

Case II. $\alpha_2, \epsilon_2 \in A_i$. Since $\epsilon_1 \notin A_i \cap E$, we have $A_i \cap E \not\supseteq A_p \cap E$. Therefore, $\mathcal{A}'$ is nonempty, so $F = A_r \cap A_s$. By essentially the same argument as in Case I, $F \subseteq A_i$. Thus the claim is proved.

The following lemma is the basis for our reduction algorithm.

LEMMA 3.4. *Let $E, A_1, \ldots, A_k$ be subsets of the ground set of T. Suppose there exists some ordering satisfying all these sets. Let $(D, F) = \text{GLUE}(E, \{A_1, \ldots, A_k\})$. Then an ordering λ satisfies these sets if and only if the following conditions hold:*

1. *λ satisfies E;*
2. *λ satisfies $E \cap A_i$ for all i;*
3. *λ satisfies $\widehat{A}_i$ for all i, where*

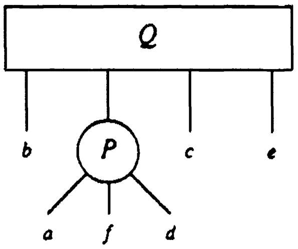


FIG. 7. *A PQ-tree is depicted. The P-nodes are indicated by circles, and the Q-nodes by rectangles. The ground set is* $\{a, b, c, d, e, f\}$, *and the frontier is* b a f d c e.

$$\widehat{A_i} = \begin{cases} A_i - E & \textit{if } A_i \textit{ and } E \textit{ have a nontrivial intersection,} \\ A_i & \textit{otherwise;} \end{cases}$$

4. λ *satisfies* D *and* F.

Proof. First, we prove the "only if" direction. Suppose λ satisfies $A_1, \ldots, A_k, E$. Then condition 1 follows trivially. Furthermore, condition 2 follows from the intersection property of Lemma 3.1 and condition 3 follows from the difference property. If D is nonempty, it is the intersection of two sets A_p and A_q that are satisfied by λ; hence by the intersection property, D is itself satisfied by λ. Similarly, F is satisfied by λ. Thus condition 4 holds.

Now we prove the "if" direction. Suppose conditions 1–4 hold of λ. Clearly, λ satisfies E. We show that λ satisfies A_i $(i = 1, \ldots, k)$.

By condition 2, λ satisfies $A_i \cap E$. By condition 3, we have that λ satisfies $\widehat{A_i}$. If A_i has a trivial intersection with E then $\widehat{A_i} = A_i$, so we are done. Assume therefore that A_i and E have a nontrivial intersection. In this case, $\widehat{A_i}$ is $A_i - E$. By condition 4, D and F are satisfied by λ. By Lemma 3.3 either $D \subseteq A_i$ or $F \subseteq A_i$. In the first case, since $A_i - E$ and $A_i \cap E$ both intersect D, it follows from the intersection property of Lemma 3.1 that λ satisfies the union $(A_i - E) \cup D \cup (A_i \cap E)$, which is just A_i. In the case where $F \subseteq A_i$, the proof is analogous. □

DEFINITION 3.5. *For a node* v *of a PQ-tree* T, $leaves_T(v)$ *denotes the set of pendant leaves of* v, *i.e., leaves of* T *having* v *as ancestor. Let* $lca_T(A)$ *denote the least common ancestor in* T *of the leaves belonging to* A. *Suppose that* $v = lca_T(A)$ *has children* $v_1 \ldots v_s$ *in order. We say* A *is* contiguous *in* T *if*

- v *is a Q-node, and for some consecutive subsequence* $v_p \ldots v_q$ *of the children of* v, $A = \bigcup_{p \le i \le q} leaves(v_i)$, *or*
- v *is a P-node or a leaf, and* $A = leaves(v)$.

For example, in Figure 7, the set $\{a, f, d\}$ is contiguous. Also, the set $\{a, f, d, c\}$ is contiguous, as is the set $\{c, e\}$. The set $\{f, d, c\}$ is not contiguous, nor is $\{b, c\}$.

The significance of contiguity is as follows. Lueker and Booth (see [32]; see also Lemma 2.1 of [29]) prove that if every ordering in $L(T)$ satisfies some set E then E is contiguous in T.

Suppose that E is indeed contiguous in T. The E*-pertinent subtree of* T *with respect to* E is the subtree consisting of $lca_T(E)$ and those children of $lca_T(E)$ whose descendents are in E. Note that the E-pertinent subtree is a PQ-tree over the ground set E. We denote this tree by $T|E$.

For a set A, define

$$A_i|E = \begin{cases} A_i \cap E & \text{if } A_i \cap E \neq E, \\ \emptyset & \text{if } A_i \cap E = E. \end{cases}$$

Remark 3.6. Suppose we modify the tree T by reducing its E-pertinent subtree with respect to a subset of E. It follows directly from the PQ-tree definitions that the result is the same as if we had reduced the whole tree T with respect to this subset. (If the reduction of the E-pertinent subtree yields the null tree, then we replace T with the null tree.)

The above observation suggest that our algorithm might profitably operate in parallel on smaller disjoint subtrees of a PQ-tree T. It is also useful to operate on a tree obtained by deleting a subtree from T.

Let $\star_E$ denote $lca_T(E)$. Let T/E denote the subtree of T obtained by omitting all the proper descendants of v that are ancestors of elements of E. Then T_0 is a PQ-tree whose ground set is $S - E \cup \{\star_E\}$. For a set A, define

$$A_i/E = \begin{cases} A_i - E \cup \{\star_E\} & \text{if } A_i \supseteq E, \\ A_i - E & \text{otherwise.} \end{cases}$$

Remark 3.7. Suppose that either $A \supseteq E$ or $A \cap E = \emptyset$. It follows from Lemma 2.18 of [29] that if we reduce T/E with respect to A/E, the effect on T is the same as if we had reduced T with respect to A. (Again, if the reduction of T/E yields the null tree, then we replace T with the null tree.)

Based on the above observations, we give a subroutine SUBREDUCE used in our algorithm MREDUCE for nondisjoint reduction. The subroutine SUBREDUCE and the main routine MREDUCE are mutually recursive. SUBREDUCE is designed in accordance with Lemma 3.4.

```
SUBREDUCE($T$, $E$, $\{A_1, \ldots, A_k\}$).
    1. Reduce $T$ with respect to $E$ using MDREDUCE.
    2. Let $(D, F) :=$ GLUE($E$, $\{A_1, \ldots, A_k\}$).
    3. If $D$ is well defined, reduce $T$ with respect to $D$ using MDREDUCE.
    4. If $F$ is well defined, reduce $T$ with respect to $F$ using MDREDUCE.
    5. In parallel, make the following recursive calls to MREDUCE:
      (a) Modify $T$ by calling MREDUCE($T|E$, $\{A_1|E, \ldots, A_k|E\}$).
      (b) Modify $T$ by calling MREDUCE($T/E$, $\{A_1/E, \ldots, A_k/E\}$).
    6. Check that in the frontier of the resulting tree $T$, each reduction set $A_i$ is consecutive.
       If so, return $T$. If not, return the null tree.
```

Each step of SUBREDUCE except for the recursive calls takes $O(\log(n + m))$ time using $n + m$ processors.

We now prove that the effect of SUBREDUCE(T, E, $\{A_1, \ldots, A_k\}$) is to reduce T with respect to $E, A_1, \ldots, A_k$. Step 1 reduces T with respect to E and A_p. Steps 3 and 4 reduce T with respect to the sets D and F. Let us assume inductively that the calls to MREDUCE correctly reduce the PQ-trees $T|E$ and T/E with respect to the given reduction sets.

In step 5(a), $T|E$ is reduced with respect to the sets $A_i|E$. Since these sets are subsets of E, as discussed in Remark 3.6, this has the effect of reducing T with respect to the same sets. If $A_i \cap E \neq E$, then $A_i|E = A_i \cap E$. Thus in this case, the effect is to reduce T with respect to $A_i \cap E$. If $A_i \cap E = E$, then $A_i|E = \emptyset$, so the reduction has no effect but reducing T with respect to E has no effect either, since T was already reduced with respect to E in step 1. Thus in either case, the effect is that of reducing T with respect to $A_i \cap E$.

In step 5(b), T/E is reduced with respect to the sets A_i/E defined immediately before Remark 3.7. If A_i has a nontrivial intersection with E, then certainly $A_i \not\supseteq E$, so $A_i/E = A_i - E$. If A_i has a trivial intersection with E, then one of the following three cases must hold: $A_i \subseteq E$, $A_i \supseteq E$, and $A_i \cap E = \emptyset$. In the first case, $A_i/E = \emptyset$, so reducing T/E

with respect to A_i/E has no effect. In the second and third cases, by Remark 3.6, the effect of reducing T/E by A_i/E is to reduce T with respect to A_i.

Thus, in general, the effect is to reduce T with respect to the set $\widehat{A}_i$ defined in condition 3 of Lemma 3.4. It follows by that lemma that if there exists an ordering $L(T)$ satisfying $E, A_1, \ldots, A_k$, then the effect of the entire call is to reduce T with respect to these reduction sets. Thus if there exists such an ordering, the resulting PQ-tree represents all such orderings. Moreover, in this case, the frontier of that PQ-tree is one such ordering, so the resulting PQ-tree is returned. Conversely, if no such ordering exists, the frontier of the PQ-tree will certainly not be such an ordering. This shows the correctness of the procedure SUBREDUCE.

The algorithm MREDUCE uses the subroutine SUBREDUCE in conjuction with a technique for choosing E, the second argument to SUBREDUCE, so that it consists of roughly half the elements of the ground set of T. This choice ensures that the recursion depth of MREDUCE is logarithmic.

Before giving the algorithm, we discuss the notion of the *intersection graph* of a collection of sets. Let $\mathcal{F}$ be a family of subsets $A_1, \ldots, A_k$ of S. The intersection graph of $\mathcal{F}$ is a graph whose nodes are the sets A_i and where two sets are considered adjacent if they intersect. In the present context, the significance of the intersection graph is given by the following easy corollary to the union property of of Lemma 3.1.

COROLLARY 3.8. *Suppose an ordering λ of S satisfies the sets $A_1, \ldots, A_k$, and the intersection graph of these sets is connected. Then λ satisfies their union $\cup_i A_i$.*

Implementation Note 4. Note that the intersection graph of $\mathcal{F}$ may have a number of edges greatly exceeding the sum of the cardinalities of the sets in $\mathcal{F}$. Therefore, to efficiently compute the connected components of the intersection graph, we construct an auxiliary bipartite graph as described in connection with the procedure GLUE. The auxiliary graph has node-set $\mathcal{F} \cup S$, and there is an edge between a set in $\mathcal{F}$ and an element of S if the element belongs to the set. Two sets in $\mathcal{F}$ are in the same connected component of the intersection graph if they are in the same component of the auxiliary graph. Moreover, a spanning forest of $\mathcal{F}$ can easily be obtained from a spanning forest of the auxiliary graph. Note that the number of edges in the auxiliary graph is just the sum of the cardinalities of the sets in $\mathcal{F}$. Thus, by using a standard connectivity algorithm [19, 43] on the auxiliary graph, we can obtain the connected components and spanning forest of the intersection graph of $\mathcal{F}$ in time $O(\log|\mathcal{F} \cup S|)$ using $|\mathcal{F} \cup S|$ processors (or $|\mathcal{F} \cup S|/\log|\mathcal{F} \cup S|$ processors using randomization).

We finally give the algorithm for multiple nondisjoint reduction.

MREDUCE(T, $\{A_1, \ldots, A_k\}$).

1. Purge the collection of input sets A_i of empty sets. If no sets remain, return.
2. Let n be the size of the ground set of T. If $n \leq 4$, carry out the reductions one by one.
3. Otherwise, let $\mathcal{A}$ be the family of (nonempty) sets A_i. Let $\mathcal{S}$ consist of the sets A_i such that $|A_i| \leq n/2$. We call such sets "small." Let $\mathcal{L}$ be the remaining, "large," sets in $\mathcal{A}$. Find the connected components of the intersection graph of $\mathcal{A}$, find a spanning forest of the intersection graph of $\mathcal{S}$, and find the intersection $\bigcap \mathcal{L}$ of the large sets.
4. Proceed according to the following four cases:

Case I. The intersection graph of $\mathcal{A}$ is disconnected. In this case, let $\mathcal{C}_1, \ldots, \mathcal{C}_r$ be the connected components of $\mathcal{A}$. For $i = 1, \ldots, r$, let E_i be the union of sets in the connected component $\mathcal{C}_i$. Call MDREDUCE to reduce T with respect to the disjoint sets $E_1, \ldots, E_r$. Next, for each $i = 1, \ldots, r$ in parallel, recursively call MREDUCE($T|E_i, \mathcal{C}_i$).

Case II. The union of sets in some connected component of $\mathcal{S}$ has cardinality at least $n/4$. In this case, from the small sets making up this large connected component, select a subset whose union has cardinality between $n/4$ and $3n/4$. (See Implementation Note 5.) Let E be this union, and call SUBREDUCE(T, E, $\{A_1, \dots, A_k\}$).

Case III. The cardinality of the intersection of the large sets is at most $3n/4$. In this case, from the large sets choose a subset of the large sets whose intersection has cardinality between $n/4$ and $3n/4$. (See Implementation Note 6.) Let E be this intersection, and call SUBREDUCE(T, E, $\{A_1, \dots, A_k\}$).

Case IV. The other cases do not hold. In this case, let E be the intersection of the large sets, and call SUBREDUCE(T, E, $\{A_1, \dots, A_k\}$).

Implementation Note 5. In this note, we address the problem arising in Case II, selecting some of the sets making up a component of size at least $n/4$. Each of these sets has size at most $n/2$, and our goal is that the union of the sets chosen has cardinality between $n/4$ and $3n/4$. A spanning tree of the component has been computed in step 3. For each of the sets comprising the component, compute the distance in the spanning tree from the root. These distances can be obtained using the Euler-tour technique [44]. Sort the sets according to distance, and let $B_1, \dots, B_s$ be the sorted sequence. Observe that any initial subsequence $B_1, \dots, B_i$ of this sequence is connected. Let $\hat{\imath}$ be the minimum i such that the union $\bigcup_{j=1}^{i} B_j$ has cardinality $\geq n/4$. Since each set is small, it follows that the cardinality of the union is no more than $3n/4$.

Implementation Note 6. In this note, we address the problem arising in Case III, selecting a subset of the large sets. Our goal is that the intersection of the selected subset of sets has cardinality between $n/4$ and $3n/4$. Order the large sets arbitrarily, and let $\hat{\imath}$ be the maximum i such that the intersection of the first i sets has cardinality at least $n/4$. Since each set has size at least $n/2$, the intersection of the first $\hat{\imath}$ has at most $n/2$ elements not appearing in the intersection of the first $\hat{\imath} + 1$ sets. The latter intersection has cardinality less than $n/4$, so the intersection of the first $\hat{\imath}$ sets has cardinality less than $3n/4$.

First, we address the correctness of the procedure MREDUCE.

LEMMA 3.9. *The PQ-tree T' returned by* MREDUCE(T, $\{A_1, \dots, A_k\}$) *satisfies*

$$L(T') = \{\lambda \in L(T) \ : \ \lambda \text{ satisfies } A_1, \dots, A_k\}.$$

Proof. Let us assume inductively that the calls to SUBREDUCE in cases II, III, and IV correctly carry out the reductions of T with respect to $E, A_1, \dots, A_k$. To verify the correctness of the call to MREDUCE in Cases II–IV, therefore, we must only check that in fact reducing T with respect to these sets is equivalent to reducing T with respect to the sets $A_1, \dots, A_k$. That is, we must prove the assertion that any ordering satisfying $A_1, \dots, A_k$ also satisfies E.

In Case II, since E is the union of a connected subcollection of the collection of reduction sets A_i, the truth of the assertion follows from the union property of Lemma 3.1. In Cases III and IV, since E is the intersection of some of the A_i's, the truth of the assertion follows from the intersection property of Lemma 3.1.

Next we address the correctness in Case I. For each component $\mathcal{C}_i$ of the intersection graph of $\mathcal{A}$, we let E_i be the union of sets in $\mathcal{C}_i$. By the union property of Lemma 3.1, any ordering satisfying $A_1, \dots, A_k$ also satisfies the sets $E_1, \dots, E_r$. Hence reducing T with respect to $A_1, \dots, A_k$ is equivalent to first reducing T with respect to $E_1, \dots, E_r$ and then reducing with respect to $A_1, \dots, A_k$. Furthermore, by Remark 3.6, reducing with respect to $A_1, \dots, A_k$ is equivalent to the reductions carried out in Case I, namely reducing each subtree $T|E_i$ with respect to the family $\mathcal{C}_i$ of sets whose union is E_i. We assume inductively that the these reductions are correctly carried out by the recursive calls to MREDUCE. This argument proves the correctness of MREDUCE in Case I. □

Now we analyze the time and processor requirements of MREDUCE.

THEOREM 3.10. *Consider an invocation* MREDUCE(T, $\{A_1, \ldots, A_k\}$). *Let n be the size of T's ground set, and let $m = \sum_i |A_i|$. The number of levels of recursion is $O(\log n)$. At each level, the sum of sizes of all ground elements in all PQ-trees is $O(n)$ and the sum of sizes of all reduction sets is $O(m)$.*

We analyze the algorithm by considering the tree R of recursive calls to MREDUCE. The root of R is the initial invocation of the procedure, and the other vertices of R are all the subsequent recursive invocations. The children of an invocation v are the invocations called by v or by an invocation of SUBREDUCE called by v. Let $T(v)$ denote the PQ-tree to which the invocation v is applied. Let $n(v)$ denote the size of the ground set of $T(v)$. Let $p(v)$ denote the parent of v in R.

The proof of Theorem 3.10 consists of three parts. In Lemma 3.11, we bound the recursion depth. In Lemma 3.12, we show that, at any level of recursion, the sum of sizes of all ground sets is $O(n)$. In Lemma 3.13, we show that at every level of recursion the sum of sizes of all reduction sets is $O(m)$.

LEMMA 3.11 (bounding the recursion depth). *The depth of recursion is $O(\log n)$, where n is the size of the ground set of the initial input PQ-tree.*

Proof. Say a vertex v is *smaller than* a vertex w if $n(v) \leq 3n(w)/4 + 1$. Consider a vertex w. If w is a Case II or Case III invocation, the choice of E in these cases is such that $n(w)/4 \leq |E| \leq 3n(w)/4$. The children of w in these cases involve the PQ-trees $T|E$ and T/E. The ground set of $T|E$ is E, so it has size at most $3n(w)/4$. The ground set of T/E is the ground set of T minus the set E, together with the element $\star_E$, so it has size at most $3n(w)/4 + 1$.

Suppose w is a Case IV invocation. The children of w involve the PQ-trees $T|E$ and T/E. In this case, E is the intersection of the large sets. Since Case III does not hold, the cardinality of E is larger than $3n(w)/4$. Hence the ground set of T/E has size at most $n(w)/4$. Thus the corresponding child is smaller than w. Consider the other child u. Its ground set is E, and its reduction sets are the sets $A_i \cap E$ that are strictly contained in E. Since E is the intersection of the large sets, only small sets A_i have the property that $A_i \cap E$ is strictly contained within E. It follows that the connected components of the reduction sets of u are contained within the connected components of the small sets of w. Since Case II does not hold, each of these connected components has size at most $n(w)/2$. Thus u is either a Case I invocation, in which case all its children are smaller than w, or the reduction sets of u form a single connected component, in which case u is a Case II invocation.

Summarizing, if w is Case II or III, then its children are all smaller than it, and if w is Case IV, then its grandchildren are all smaller than it. Finally, if w is a Case I invocation, none of its children is a Case I invocation, so its great-grandchildren are all smaller than it. We infer that the number of levels of recursion is $O(\log n)$. □

LEMMA 3.12 (bounding the sum of sizes of ground sets). *For any level of recursion, the sum of sizes of ground sets of all PQ-trees being recursed on is $O(n)$, where n is the size of the ground set of the initial input PQ-tree.*

Proof. We show that the ground sets of all PQ-trees at a given level of recursion are disjoint and that all but n of the elements are in the ground set of the initial input PQ-tree.

For any vertex w that is Case I, the ground sets of the children of w are disjoint subsets of the ground-set of w. Hence no new ground-set elements are introduced by a Case I vertex. Each vertex w that is Case II, III, or IV has two children, one working on $T(w)|E$ and one working on $T(w)/E$. The ground set of the first child is E, a subset of the ground set of $T(w)$, and that of the second is the ground set of T minus the set E, together with a new element $\star_E$.

We see that every vertex's children have disjoint ground sets. It follows by induction on j that the ground sets of all PQ-trees at level j of the recursion tree R are disjoint.

We have also seen that each vertex introduces at most one new ground set element and that only vertices with two children introduce such new elements. Furthermore, each vertex has a nonempty ground set. Let R' denote the recursion tree R truncated at some level j. Let s be the number of ground-set elements at that level that do not belong to the ground set of the initial input PQ-tree. Then the total number of ground set elements is $n + s$. Since each PQ-tree has a nonempty ground set and the ground sets of the leaves of R' are all disjoint, the number of leaves of R' is at most $n + s$. Hence the number of internal vertices having two or more children is at most $(n + s)/2$. The number of new ground elements is at most the number of internal vertices having two or more children, so $s \leq (n + s)/2$. It follows that $s \leq n$. □

LEMMA 3.13 (bounding the sum of sizes of all reduction sets). *At any level of recursion, the sum of sizes of all reduction sets is $O(m)$, where m is the sum of sizes of reduction sets in the initial invocation of* MREDUCE.

Proof. We analyze the way reduction sets for one level of recursion are transformed by SUBREDUCE into reduction sets at the next level of recursion. There is a forest that represents this process. The vertices of the forest are pairs (invocation, reduction set). Consider one invocation $u = \text{MREDUCE}(T, \{A_1, \ldots, A_k\})$. Depending on which case arises during this invocation, each reduction set A_i gives rise to one or two reduction sets in child invocations. In Case I, each reduction set A_i gives rise to one reduction set, namely A_i, in some child invocation v. In this case, the only child of (u, A_i) is (v, A_i). In Cases II–IV, each reduction set A_i gives rise to two, $A_i|E$ and A_i/E. In these cases, the children of (u, A_i) are the pairs $(v, A_i|E)$ and $(w, A_i/E)$, where v and w are the appropriate children of the invocation u. Note that the reduction set A_i gives rise to disjoint reduction sets. Moreover, in Cases II–IV, a new element $(\star_E)$ may be included in A_i/E. Thus each vertex introduces at most one new element, and only vertices with two children introduce such new elements.

The remainder of the proof is similar to that of Lemma 3.12. Let Q' be the forest Q truncated at some recursion level j. We focus on the reduction sets in the leaves of Q'. Let s be the number of occurences in the leaf reduction sets of elements not belonging to the reduction sets of the original invocation. Then the sum of sizes of all the leaf reduction sets is $m + s$. Since empty reduction sets are discarded, we may assume that every leaf of Q' has a nonempty reduction set. Thus the number of leaves is at most $m + s$. Hence the number of internal vertices with two children is at most $(m + s)/2$. As in the proof of Lemma 3.12, it follows that $s \leq m$. □

This completes the proof of Theorem 3.10. Since each level of MREDUCE takes $O(\log(n + m))$ time and the recursion depth is $O(\log n)$, the total time required is $O(\log n \log(n + m))$ using $O(n + m)$ processors.

4. Applications.

In this section, we show how having a PEO for a chordal graph enables one to solve many problems efficiently in parallel. The key to our efficient algorithms is our use of the *elimination tree*. The elimination tree is a structure introduced by [42] in the context of sparse Gaussian elimination but implicit in the work of others, including [40]. In §4.1, we show that the elimination tree determined by a PEO of a chordal graph has useful separation properties. Most of the chordal-graph algorithms described in this chapter rely on the elimination tree.

4.1. The elimination tree determined by a PEO.

Let \$ be a one-to-one numbering of the nodes of the connected graph G. As in §2, we shall say v is *richer* than u and u is *poorer* than v if the number assigned to v is higher than that assigned to u. We define the *elimination tree* $T(G_\$)$ of $G_\$$ as follows. For every node v except the highest numbered, v's parent $p(v)$

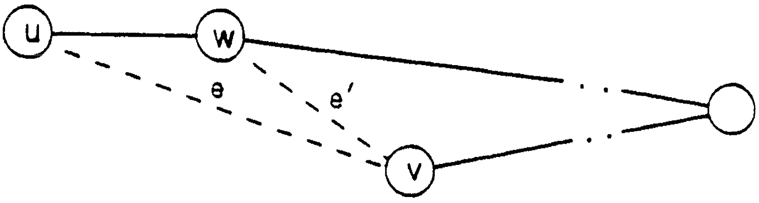


FIG. 8. *The existence of a cross-edge e connecting u and v implies the existence of a cross-edge e′ connecting w and v.*

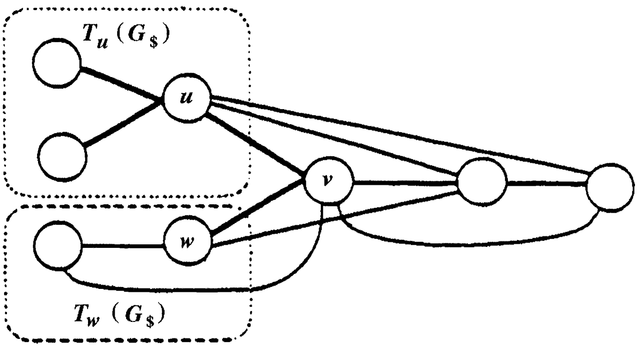


FIG. 9. *When the node v and its richer neighbors are removed, the subtrees rooted at children of v become separated from each other and from the remainder of the graph.*

is defined to be the poorest neighbor of v that is richer than v. The tree $T(G_\$)$ can easily be constructed from $G_\$$ in $O(\log n)$ time using $(n+m)/\log n$ processors.

Since parents are richer than their children, there are no directed cycles in $T(G_\$)$. Since each vertex (except the richest) has exactly one parent, $T(G_\$)$ is in fact a tree. Recall that a PEO of a chordal graph is a numbering of the nodes of the graph such that for each node v, the richer neighbors of v form a clique. Define a *cross-edge* to be an edge of G such that neither endpoint is an ancestor of the other in $T(G_\$)$. If there are no cross-edges, we call $T(G_\$)$ a *depth-first search tree.* If \$ is a PEO, the existence of a cross-edge between the node u and a poorer node v implies the existence of a cross-edge between u and the parent of v; using induction on the distance in the tree between endpoints of an edge, we can prove the following lemma.

LEMMA 4.1. *Let G be a chordal graph. If* \$ *is a PEO of G, then* $T(G_\$)$ *has no cross-edges.*

Proof. Let u and v be two nodes; we show that there is no cross-edge between u and v by induction on the length of the path in $T(G_\$)$ connecting u and v. If this length is 1, v is the parent of u or vice versa, so an edge between them is not a cross-edge. Therefore, assume the length is greater than 1.

Suppose e is an edge between u and v. Assume without loss of generality that v is richer than u. Let w be the parent of u; by choice of parent, w is richer than u but poorer than v. See Figure 8. Using e, we can form a backward path through u with one endpoint w and the other v, proving via the Backward-Path Theorem the existence of an edge e' between w and v. By the inductive hypothesis, e' is not a cross-edge, so v must be an ancestor of w in $T(G_\$)$. This shows that e is not a cross-edge. □

We next show $T(G_\$)$ has desirable separation properties. For a node v, let $T_v(G_\$)$ denote the subtree of $T(G_\$)$ rooted at v. As illustrated in Figure 9, removing v and its richer neighbors separates the subtrees rooted at children of v from the remainder of the graph.

LEMMA 4.2. *Let \$ be a PEO of G. Let v be a node of G with children $v_1, \ldots, v_k$ in $T(G_\$)$. Let K be the clique of G consisting of v and its richer neighbors. Then $G[T_{v_i}(G_\$)]$ is a connected component of $G - K$, for $i = 1, \ldots, k$.*

Proof. To see that $G[T_{v_i}(G_\$)]$ is connected in G, note that edges in $T_{v_i}(G_\$)$ are edges in G, and hence $T_{v_i}(G_\$)$ is a spanning tree of $G[T_{v_i}(G_\$)]$. None of the nodes in $T_{v_i}(G_\$)$ are in K, so $G[T_{v_i}(G_\$)]$ remains connected when K is removed from G.

Suppose there is an edge between a node v' in $T_{v_i}(G_\$)$ and a node w not in $K \cup T_{v_i}(G_\$)$. The edge cannot be a cross-edge in $T(G_\$)$, so w must be an ancestor of v'; since w is not in $T_{v_i}(G_\$)$, it must be an ancestor of v as well. Using the edge, we can construct a backward path from w through v' and up the tree $T(G_\$)$ to v. By the Backward-Path Theorem, w must be adjacent to v, so w belongs to K, a contradiction. □

As a corollary to Lemma 4.2, we can show that a chordal graph has a clique whose removal breaks the graph into pieces of at most half the size. (This fact was first shown in [21].) Let the node v of Lemma 4.2 be the lowest node in the elimination tree having more than $n/2$ descendents. Then every component $G[T_{v_i}(G_\$)]$ has at most $n/2$ nodes, but together these components comprise at least $n/2$ nodes. Hence the clique consisting of v together with its richer neighbors forms a separating clique. We use this idea below in our algorithm for finding an optimal coloring.

4.2. Recognition. A recognition algorithm for chordal graphs follows easily from the PEO algorithm. When the PEO algorithm produces a total ordering \$ of G, the correctness of the algorithm implies that if G is chordal then \$ is a PEO; of course, if \$ is a PEO, then G is chordal. It therefore suffices to check whether \$ is a PEO. We can parallelize a technique used in [40]. Each node v sends to its parent $p(v)$ in $T(G_\$)$ a list of v's richer neighbors (excluding $p(v)$). Then each node w sorts the elements of all the lists it received, together with w's own adjacency list, and verifies that it is a neighbor of every node on every list it received.

CLAIM. *The numbering \$ is a PEO if and only if no verification step fails.*

Proof. Suppose \$ is a PEO. Then for every node v, the richer neighbors of v form a clique. In particular, the parent of v is adjacent to all v's other richer neighbors. Thus every verification step succeeds.

Suppose no verification step fails. We claim that for each node v, the richer neighbors of v form a clique. The proof is reverse induction on the depth d of v in the tree $T(G_\$)$. The claim is trivial for $d = 0$, because the root has no richer neighbors. Suppose the claim holds for d, and let v be a node at depth $d + 1$. By the inductive hypothesis, $p(v)$ and its richer neighbors form a clique K. By the success of $p(v)$'s verification step, every richer neighbor of v is a neighbor of $p(v)$ and hence lies in K. This proves the induction step. □

The claim shows that we can determine whether a given ordering is a PEO of G. The time for carrying out verification is $O(\log n)$ using $n + m$ processors. For subsequent applications, assume that the numbering \$ is a PEO of the chordal graph G.

4.3. Maximum-weight clique. Fulkerson and Gross observed that every maximal clique S of G is of the form

$$\{v\} \cup \{\text{richer neighbors of } v\}.$$

To see this, we need only let v be the poorest node of S. It follows that the maximal cliques of G can be determined from \$. Suppose each node is assigned a nonnegative weight. As Gavril observed, any maximum-weight clique is maximal, so a maximum-weight cliques may easily be determined from \$.

4.4. Depth-first and breadth-first search trees. We showed in §4.1 that the elimination tree determined by a PEO \$ is a depth-first search tree. To obtain a breadth-first search tree of G, we construct a tree similar to the elimination tree by choosing the parent of each node v (except the richest node) to be the richest neighbor of v. Let T be the resulting tree, rooted at the richest node, which we shall denote by r. Our proof that T is a breadth-first search tree relies on two claims.

CLAIM 4.3. *For each node v, the shortest path from v to r in G is monotonically increasing in wealth.*

Proof. Any subpath whose internal nodes are poorer than its endpoints can be replaced by a direct edge between the endpoints, by the validity of \$. □

For the second claim, let $d(v)$ denote the length of the shortest path in G from v to r.

CLAIM 4.4. *If w is a descendent of v in the elimination tree, then $d(w) \geq d(v)$.*

Proof. The proof is by reverse induction on the wealth of w. The basis, in which $w = r$, is trivial. Otherwise, let w' be the second node on a shortest path in G from w to r, so $d(w) = 1 + d(w')$. By Claim 4.3, w' is richer than w and hence an ancestor of w in the elimination tree by Lemma 4.1. If w' is a descendent of v in the elimination tree, then $d(w') \geq d(v)$ by the inductive hypothesis. If w' is an ancestor of v, then there is a backward path from v back along tree edges to w and then forward to w', proving by validity of \$ that v is adjacent to w' in G and hence that $d(v) \leq 1 + d(w')$. □

For each node $v \neq r$, let $p(v)$ be the richest neighbor of v in G. Any other neighbor w of v is a descendent of $p(v)$, so $d(w) \geq d(p(v))$ by Claim 4.4. It follows that $p(v)$ is the second node in a shortest path in G from v to the richest node of G. Thus the tree defined by $p(\cdot)$ is a breadth-first search tree.

4.5. Maximum independent set.

Gavril showed that a maximum independent set $\mathcal{I}$ of the chordal graph G is obtained by the greedy maximal-independent-set algorithm when applied to nodes in order of the PEO $\$ = v_1 \ldots v_n$. His algorithm proceeds as follows. First, put v_1 into $\mathcal{I}$, and delete v_1 and its neighbors. Next, put the poorest remaining node in $\mathcal{I}$, and so forth. Once $\mathcal{I}$ has been found, the family of cliques of the form $\{x\} \cup \{\text{richer neighbors of } x\}$ for $x \in \mathcal{I}$ is a clique cover (a set of cliques whose union contains all the nodes). Because any independent set has size at most that of any clique cover, it follows that the above procedure has identified a *maximum* independent set and a *minimum* clique cover.

We want to simulate Gavril's sequential greedy algorithm in parallel. First, suppose that the elimination tree $T(G_\$)$ is a path with leaf x. In this case, we give a simple algorithm PMIS for simulating Gavril's algorithm. For each node v, let $b[v]$ be the lowest ancestor of v in $T(G_\$)$ that is not adjacent to v in G (or v if no such ancestor exists).

CLAIM 4.5. *The greedy independent set consists of x, $b[x]$, $b[b[x]]$, and so on.*

This set can be determined quickly in parallel using standard pointer-jumping techniques. The implementation shown in Figure 10 requires $O(\log n)$ time, m processors, and $O(m \log n)$ space; the use of more sophisticated techniques (e.g., [3], [10]) achieves the same time bound using only $m/\log n$ processors and $O(m)$ space.

Proof. Suppose we put x into $\mathcal{I}$ and delete the neighbors of x. The node $b[x]$ is by definition the poorest undeleted node. Moreover, we assert that for each undeleted node v, $b[v]$ is undeleted. If $b[v]$ were a neighbor of x, then there would be a backward path from $b[v]$ back to x and then up the tree to v; thus v would be adjacent to $b[v]$ by the Backward-Path Theorem, contradicting the definition of $b[v]$. This argument proves the assertion. The claim follows by induction on the length of the elimination path. □

To generalize this procedure to the case in which $T(G_\$)$ is a tree, we use an idea of Naor, Naor, and Schäffer: eliminating terminal branches. A *terminal branch* of a tree is a maximal path of degree-two nodes ending in a leaf. Naor, Naor, and Schäffer observe that deletion of all terminal branches of a tree yields a new tree with half as many leaves. Therefore, $O(\log n)$

```
PMIS
P1   For each node $v$, let $b_0[v]$ denote the lowest ancestor of $v$ that is not adjacent to $v$ (or else $v$).
P2   For stages $k = 0, \ldots, \lceil \log n \rceil - 1$, for each node $v$, let $b_{k+1}[v] := b_k[b_k[v]]$.
P3   Mark the leaf $x$ as being in the independent set.
P4   For stages $k = \lceil \log n \rceil - 1, \lceil \log n \rceil - 2, \ldots, 0$, for each marked node $v$, mark $b_k[v]$.
```

FIG. 10. *A simple implementation of the algorithm* PMIS *for finding a maximum independent set when the elimination tree is a path.*

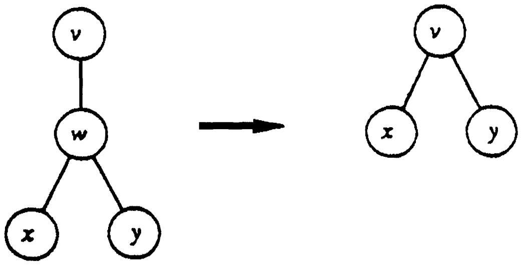


FIG. 11. *To splice a node out of a tree, remove the node and reattach the node's children to its parent.*

iterations of terminal branch elimination suffice to eliminate the entire tree. They apply this idea to the *clique tree* of a chordal graph, a tree representing the structure of intersections among maximal cliques, in order to obtain a parallel algorithm for finding a maximal independent set. However, even assuming the clique tree is given, they prove only that the number of processors required is $O(n^2)$. By applying the idea to the elimination tree, we obtain a simpler algorithm requiring only $m/\log n$ processors.

Before giving the algorithm, we introduce a bit of tree surgery, called *splicing*. To *splice* a node v out of a tree T is to remove the node and reattach any children of v to v's parent in T, as illustrated in Figure 11. If a set of nodes are to be spliced from a tree, the resulting tree does not depend on the order in which the nodes are spliced out. In fact, they can all be spliced out at once; for each node v to be spliced out, the children of v are reattached to the lowest ancestor of v that is not spliced out.

We now describe the algorithm MIS, shown in Figure 12, for constructing a maximum independent set in a chordal graph G. The algorithm maintains a set $\mathcal{I}$, the independent set under construction, and a tree T, obtained from the elimination tree $T(G_\$)$ by splicing out nodes. We prove by induction that the following invariant holds before and after each iteration of the algorithm.

Invariant.

(1) $\mathcal{I}$ is an independent set.

(2) Every neighbor of a node of $\mathcal{I}$ is in fact a *richer* neighbor of some node of $\mathcal{I}$.

(3) T is obtained from $T(G_\$)$ by splicing out the nodes of $\mathcal{I}$ and their neighbors.

The algorithm terminates when T is empty, at which point $\mathcal{I}$ is an independent set such that every node of G is either in $\mathcal{I}$ or a richer neighbor of some node in $\mathcal{I}$. Thus, as in Gavril's algorithm, the family of cliques of the form $\{x\} \cup \{\text{richer neighbors of } x\}$ for $x \in \mathcal{I}$ is a minimum clique cover, and $\mathcal{I}$ is a maximum independent set.

Initially, $\mathcal{I} = \emptyset$ and $T = T(G_\$)$, so the invariant holds trivially. Suppose the invariant holds through the first k iterations of the algorithm, and consider the $k+1$st iteration. For each

```
                                   MIS
M1   To initialize, let $\mathcal{I} = \emptyset$ and let $T = T(G_\$)$.
M2   While $T$ is not empty,
M3       Use the algorithm PMIS to find the greedy maximum independent set $\mathcal{I}_\mathcal{B}$ of the subgraph
         induced on each terminal branch $\mathcal{B}$ of $T$.
M4       Add the nodes $\bigcup_\mathcal{B} \mathcal{I}_\mathcal{B}$ to $\mathcal{I}$.
M5       Splice out of $T$ the nodes
         $\bigcup_\mathcal{B}(\mathcal{I}_\mathcal{B} \cup \{\text{neighbors of } \mathcal{I}_\mathcal{B}\})$.
```

FIG. 12. *The algorithm* MIS *to construct a maximum independent set $\mathcal{I}$ in the chordal graph G.*

terminal branch $\mathcal{B}$, the algorithm finds a maximum independent set $\mathcal{I}_\mathcal{B}$ of the subgraph induced on B, using PMIS as a subroutine. If two nodes of T lie in different terminal branches, neither is an ancestor of the other in $T(G_\$)$, and hence the two nodes are not adjacent, by Lemma 4.1. Thus $\bigcup_\mathcal{B} \mathcal{I}_\mathcal{B}$ is an independent set in G, where the union is over all terminal branches of T. Moreover, T contains no neighbors of $\mathcal{I}$ (by part (3) of the invariant), so $\mathcal{I} \cup (\bigcup_\mathcal{B} \mathcal{I}_\mathcal{B})$ is an independent set of G. Thus when the nodes $\bigcup_\mathcal{B} \mathcal{I}_\mathcal{B}$ are added to $\mathcal{I}$ in step M4, part (1) of the invariant remains true.

To show that part (2) remains true, we must prove that every node w that is newly a neighbor of a node in $\mathcal{I}$ is in fact a richer neighbor of a node in $\mathcal{I}$. Our simulation PMIS of Gavril's algorithm on terminal branches $\mathcal{B}$ guarantees this property when w lies on a terminal branch. Suppose, therefore, that w does not lie on a terminal branch, and let $v \in \mathcal{I}$ be a neighbor of w. By Lemma 4.1, v is either a descendent or an ancestor of w in $T(G_\$)$. The set $\mathcal{I}$ consists only of nodes in terminal branches of T and descendents of such nodes. Hence v must be a descendent of w and also a poorer neighbor.

In the last step of an iteration of the algorithm, we splice out of T all nodes newly added to $\mathcal{I}$ and their neighbors. This step ensures that part (3) holds at the end of the iteration.

Having proved that the invariant continues to hold, we now consider the implementation of the algorithm MIS. In step M3, the algorithm must identify the nodes lying in terminal branches of T. An application of the Euler-tree technique [44] suffices to determine, for each node v of T, the number of leaf descendents of v. The nodes for which this number is 1 are the nodes in terminal branches. Next, the algorithm must find a maximum independent set in each terminal branch. For each node v in T, $b_0[v]$ is assigned the lowest ancestor w of v in T that is not a neighbor of v, if w lies in a terminal branch. Otherwise, $b_0[v]$ is assigned v. As in PMIS, a pointer-jumping technique is then used to mark the nodes $x, b_0[x], b_0[b_0[x]]$, and so on, for all leaves x of T. The marked nodes are added to $\mathcal{I}$ in step M4.

To implement the splicing in step M5, we again use a pointer-jumping technique; for each node v, we compute the lowest ancestor of v in T that is not to be spliced out. Each step can be implemented in $O(\log n)$ time using $m/\log n$ processors and $O(m)$ space. Each iteration removes all nodes in terminal branches and hence reduces the number of leaves in T by a factor of two; consequently, $\lceil \log n \rceil + 1$ iterations suffice, for a total of $O(\log^2 n)$ time.

4.6. Optimal coloring. Gavril showed that applying the greedy coloring algorithm to the nodes of G in reverse order of \$ yields an optimal coloring. Our basic approach to coloring the graph in parallel is as follows: choose a clique K such that the components $H_1, \ldots, H_s$ of $G[G - K]$ are all "small," recursively color each subgraph $G[H_i \cup K]$, repair the colorings by making them consistent on the nodes of K, and merge the repaired colorings.

Naor, Naor, and Schäffer use essentially this approach in their coloring algorithm. Their algorithm, however, uses n^3 processors even if all maximal cliques are provided. One apparent

```
COLOR($G$, $K_0$).
Input: Connected graph $G$ containing a clique $K_0$, such that every node of $K_0$ has a neighbor in $G - K_0$.
Output: Optimal coloring of $G$.
C1    If $G - K_0$ consists of a single node $v$, then $G$ is a clique; assign the first $|V(G)|$ colors to
      its nodes, and end.
C2    Break $G - K_0$ into subgraphs $H_0, \ldots, H_s$ such that
        • each subgraph has size at most half that of $G - K_0$;
        • $H_1, \ldots, H_s$ are distinct components of $G - K_0 - H_0$; and
        • for $1 \leq i \leq s$, the neighborhood of $H_i$ in $G - H_i$ is a clique $K_i$.
C3    For $i = 0, \ldots, s$ in parallel,
      call COLOR($\widehat{H}_i$, $K_i$), where $\widehat{H}_i = G[H_i \cup K_i]$, to get an optimal coloring $c_i$ of $\widehat{H}_i$.
C4    For $i = 1, \ldots, s$ in parallel, modify the coloring $c_i$ to be consistent with $c_0$ on the nodes
      $V(K_i)$ they have in common.
C5    Merge the colorings to obtain a coloring of $G$.
```

FIG. 13. *The recursive algorithm* COLOR *for finding an optimal coloring of a chordal graph G.*

difficulty is that the subgraphs on which the algorithm recurs are not disjoint—they share the nodes in K—so we cannot hope to make do with only one processor per edge.

In coping with this difficulty, we use the same idea that made our PEO algorithm efficient. Given the knowledge that K is a clique, we need not inspect the edges between nodes of K during the recursive calls. We recursively solve the following problem: given a chordal graph G and a clique K_0 contained in G, find an optimal coloring of G. We solve this problem in $O(\log t \log |G - K_0|)$ time using t processors, where t is the number of edges with at least one endpoint in $G - K_0$, or using $t/\log t$ processors of a randomized PRAM. The algorithm, COLOR(G, K_0), is shown in Figure 13. There are $1 + \log |G - K_0|$ levels of recursion. We shall show that each level can be implemented in $O(\log t)$ time. To find a coloring in the original graph G, we call COLOR(G, $\emptyset$).

Step C4, in which we modify the colorings to be consistent, can be implemented using a parallel prefix computation. We shall give more details later. The idea in implementing step C2, in which we divide up the graph, is as follows. We inductively assume we have an elimination tree $T(G_\$)$ in which the nodes of K_0 are the richest nodes. Using the Euler-tour technique [44], choose the lowest node $\hat{v}$ in $T(G_\$)$ that has more than $p/2$ descendents, where $p = |G - K_0|$. Let $v_1, \ldots, v_s$ be the children of $\hat{v}$ in $T(G_\$)$; then each subtree $T_{v_i}(G_\$)$ has at most $p/2$ nodes. We let K be the clique $\{\hat{v}\} \cup \{\text{richer neighbors of } \hat{v}\}$ and let $H_i = G[T_{v_i}(G_\$)]$. By Lemma 4.2, the subgraphs $H_1, \ldots, H_s$ are connected components of $G - K$. The neighborhood of each H_i in $G - H_i$ is contained in the clique K and is therefore itself a clique K_i. Let $H_0 = G - K_0 - \bigcup_{i=1}^{s} H_i$. By choice of $\hat{v}$, H_0 has at most $p/2$ nodes.

We shall inductively ensure that for $i = 0, \ldots s$,

(I) c_i is an optimal coloring of $G[H_i \cup K_i]$;

(II) the maximum color used by c_i equals the number of colors used; and

(III) the coloring c_i assigns the first $|K_i|$ colors to the nodes of K_i.

These conditions are easy to establish at the base of the recursion, step C1. Condition (III) is automatically preserved in going from one level of recursion to the next higher level: the colors assigned by c to the nodes of K_0 are exactly those assigned by c_0. Assume (I), (II), and (III) hold for $c_0, \ldots, c_s$. We must ensure that (I) and (II) hold for the coloring c of G that we construct. Namely, we must color G with colors 1 through x, where x is the minimum number of colors needed to color G. We shall, in fact, construct a coloring c of G with maximum color equal to

$$(1) \qquad \max\{\text{number of colors used by } c_i \ : \ i = 0, \ldots, s\}.$$

Since c_i is an optimal coloring of the subgraph $G[H_i \cup K_i]$ of G, the value given by (1) is clearly a lower bound on x. Thus by achieving this lower bound, we ensure that our coloring c of G is optimal.

The colors c assigns to nodes of $H_0 \cup K_0$ are exactly the colors c_0 assigns to these nodes. Therefore, for any node $v \in H_0 \cup K_0$, the color assigned to v is no more than the value of (1). It remains to determine the colors c assigns to nodes of H_i, for $i = 1, \ldots, s$.

Let x_i be the number of colors used by c_i. The colors 1 through $|K_i|$ are assigned by c_i to the nodes of K_i. The nodes of H_i are asigned colors $|K_i| + 1$ through x_i. For each q, $|K_i| + 1 \leq q \leq x_i$, let

$$A_i[q] := |\{c_0(v) < q \ : \ v \in V(K_i)\}|.$$

(The values $A_i[\cdot]$ can be computed using a parallel prefix computation.) For each node $v \in H_i$, define

$$c(v) := c_i(v) - |K_i| + A_i[c_i(v)].$$

Thus the colors of nodes of H_i are remapped to colors starting at 1, with gaps only for colors already assigned to nodes of K_i. This assignment ensures that, for any node $v \in H_i$, colors 1 through $c(v)$ all appear in the coloring c_i' induced by c on H_i. Since c_i' can be obtained from c_i by merely permuting colors, it follows that $c(v)$ is no more than the value of (1). We have completed the proof of correctness of the algorithm.

The only nontrivial computation in implementing step C4 is computing the $A_i[q]$ values. For each $1 \leq i \leq s$, we identify the colors c assigns to nodes of K_i and then perform a parallel prefix computation of length $x_i \leq |H_i|$. The total work is proportional to $\sum_i |K_i| + |H_i|$.

Let t_i be the number of edges that either lie in H_i or connect H_i to K_i. Since H_i is connected, the number of nodes in H_i is at most one more than the number of edges in H_i. Since every node in K_i is a neighbor of some node in H_i, the number of nodes in K_i is at most the number of edges connecting H_i to K_i. Thus $|H_i \cup K_i| \leq t_i + 1$, so the total work is proportional to $\sum_i t_i$, which is just the number t of edges that either lie within $G - K_0$ or connect $G - K_0$ to G.

Assume inductively that $O(\log t_i \log |H_i|)$ time and $O(t_i / \log t_i)$ processors are sufficient to recursively color $G[H_i \cup K_i]$. Then $O(t/\log t)$ processors are sufficient to recursively color all the subgraphs $G[H_i \cup K_i]$ in $O(\log t(\log(|G - K_0|) - 1))$ time and to combine the colorings in $O(\log t)$ time, for a total of $O(\log t \log |G - K_0|)$ time.

4.7. PQ-tree intersection. We can also test two leaf-labelled PQ-trees for isomorphism. The idea is to use Edmonds' tree-isomorphism algorithm (see [1]), which proceeds in stages from the leaves to the roots, level by level. In general, the number of levels may be large, so we instead apply Edmonds' algorithm to *decomposition trees* for the original PQ-trees. The decomposition tree of a tree T is formed by breaking T into subtrees of half the size by removing the edges from a node v to its children, recursively finding decomposition trees of the subtrees, and hanging the recursive decomposition trees from a common root. By labeling the decomposition tree during its construction, one can ensure that it uniquely represents T up to isomorphism. The decomposition trees for n-leaf PQ-trees have $O(\log n)$ levels, so

$O(\log n)$ stages of Edmonds' algorithm suffice. Each stage involves sorting strings, which can be done in $O(\log n)$ time on a "priority" type CRCW PRAM using Cole's algorithm [9], for a total time bound of $O(\log^2 n)$. To achieve this time bound, $(n+t)/\log n$ processors are sufficient, where t is the sum of the lengths of the leaf labels.

4.8. Interval graphs. The algorithm of Booth and Lueker for recognition of interval graphs is as follows: Find a PEO of the input graph G; if there is none, the graph is certainly not an interval graph. Otherwise, we can obtain all the maximal cliques (there are at most n; see §4.3). For each node v, let A_v be the set of maximal cliques containing v. It can be shown [17] that $\sum_v |A_v| = O(m+n)$. Let T be the universal PQ-tree whose ground set is the set of maximal cliques. Reduce T with respect to the sets A_v. If the resulting PQ-tree T_G is the null tree, then it follows from a theorem of Gilmore and Hoffman [22] that G is not an interval graph. Otherwise, an ordering represented by T yields a representation of G as an intersection of intervals.

Since we have given efficient parallel algorithms for finding a PEO and for PQ-tree multiple (nondisjoint) reduction, the above algorithm is parallelizable; each step takes $O(\log^2 n)$ time using $O(n+m)$ processors.

Booth and Lueker showed (see [32]; also see [8]) that if the PQ-tree T_G derived from G is augmented with some labels depending on the graph, isomorphic interval graphs correspond to isomorphic labeled PQ-trees. Booth and Lueker then showed that such labeled PQ-trees could be tested for isomorphism in linear time, proving that interval-graph isomorphism testing could be done in linear time. We show that this approach can be parallelized.

Let T_G be the PQ-tree for an interval graph G. We augment T_G with labels as follows: Each leaf x of T_G corresponds to a maximal clique $\mathcal{C}_x$; we create a label for x by sorting the degrees of the nodes in $\mathcal{C}_x$. The sum of the lengths of the strings labeling T_G is just the sum of the sizes of the maximal cliques, which is $O(n+m)$. The labels can be found in $O(\log n)$ time using $O(n+m)$ processors by means of small-integer sorting. It follows from Theorem 1 of [32] and the proof of Lemma 3.1 of [8] that the resulting labeled PQ-tree uniquely represents the interval graph G up to isomorphism.

The number of nodes in the augmented PQ-tree is $O(m+n)$, and the augmentation can easily be carried out in parallel. It remains to show how such augmented PQ-trees may be tested for isomorphism in $O(\log^2 n)$ time using $m+n$ processors. To achieve this time bound, we require a powerful model of parallel computation, the "priority" CRCW PRAM. Multiple processors are permitted to write to the same location in the same time step; the value stored in the location is the value written by the lowest-numbered processor.

The standard (sequential) approach to tree isomorphism (see [1], [26]) and the approach used in [32] is to process the two trees from the leaves up, essentially canonicalizing subtrees at each successive level by sorting. The problem with a direct parallelization of this approach is that there may be too many levels. Therefore, our approach is to test isomorphisms not of the original trees but of their decomposition trees, which are guaranteed to have only a logarithmic number of levels.

In constructing a decomposition tree for an augmented PQ-tree, we must for the recursion construct decomposition trees for slightly more general trees: trees that are obtained from augmented PQ-trees by deleting some subtrees and assigning numbers to some (resulting) leaves. Let T be such a modified PQ-tree. Note that, for example, T may contain P-nodes and Q-nodes that have no children. The leaves of the decomposition tree for T will correspond to the nodes of T.

We form T's decomposition tree $\mathcal{D}_j(T)$ as follows: if T consists of a single node x, then $\mathcal{D}_j(T)$ also consists of a single vertex v. The vertex v is labeled with P, Q, L, or D, depending on whether x is a P-node, a Q-node, a leaf, or a degree node in the original augmented PQ-tree. Moreover, if x was numbered, then v is labeled with the same number.

Suppose T contains at least two nodes. There is a unique node $\hat{x}$ in T that is the lowest node of T that has more than $n/2$ descendents (including itself). The choice of $\hat{x}$ is invariant under automorphisms of T. Moreover, $\hat{x}$ is not a leaf. Removal of the edges joining $\hat{x}$ to its children disconnects T into subtrees each having at most $n/2$ nodes. Let these subtrees be $T_0, \ldots, T_k$, where T_0 is the subtree containing $\hat{x}$, and $T_1, \ldots, T_k$ are ordered just as their roots are ordered as children of $\hat{x}$ in T. Modify T_0 by assigning the number of nodes of T to the leaf $\hat{x}$. Then $\mathcal{D}(T)$ is defined to be the tree obtained from $\mathcal{D}(T_0), \ldots, \mathcal{D}(T_k)$ by introducing a new vertex v to be the parent of the roots $v_0, \ldots, v_k$ of these $k+1$ decomposition trees, in this order. The vertex v is labelled with P, Q, L, or D as before.

The assignment of integers to leaves of the modified PQ-trees and to leaves of the decomposition trees makes it possible to uniquely reconstruct a PQ-tree T from its decomposition tree $\mathcal{D}(T)$, while at the same time establishing the correspondence between the nodes of T and the leaves of $\mathcal{D}(T)$. Let v be the root of $\mathcal{D}(T)$. Recursively reconstruct the modified PQ-trees whose decomposition trees are rooted at the children of v in $\mathcal{D}(T)$. Let $T_0, \ldots, T_k$ be the resulting PQ-trees in order. There is a unique leaf $\hat{x}$ in T_0 labeled with the integer $\sum_i = 0^k |T_i|$. To construct T, let the roots of $T_1, \ldots, T_k$ be the children of $\hat{x}$ in order, and remove the label from $\hat{x}$. It follows that two decomposition trees are isomorphic if and only if their decomposition trees are isomorphic.

It remains to describe how to test decomposition trees for isomorphism. The approach used is essentially the standard approach, but we must take care to respect the P, Q, L, D labels, the integer labels assigned during augmentation, the integer labels assigned during decomposition, and the order of children of Q-nodes. It follows from the construction of the decomposition trees that only leaves have integer labels.

The isomorphism algorithm proceeds top to bottom, from the leaves of the decomposition trees to their roots. The *height* of a vertex in a tree is defined to be the maximum distance down the tree to a leaf. We initialize by assigning a string to each leaf (height-0 vertex) of each tree. The string combines all the labels of the leaf.

For the general step of the algorithm, we are given an assignment of strings to the height-h vertices of the trees. We sort these strings, eliminate duplicates, and assign to each string the ordinal number of the string in the set of strings. If the multiset of strings associated with height-h vertices of one tree does not match the corresponding multiset of the other tree, we terminate the algorithm because the trees are not isomorphic. We also terminate the algorithm if one tree has height-$(h+1)$ vertices and the other does not.

Otherwise, if neither tree has height-$(h+1)$ vertices, we conclude that the trees are isomorphic, and if both do, we next assign strings to these vertices as follows: let v be a height-$(h+1)$ vertex. It follows from the construction of the decomposition tree that v corresponds to an internal node of the original augmented PQ-tree and is therefore labeled with P, Q, or L. If v is labeled with P or L, we form its string as follows: the first element of the string is either P or L, whichever is appropriate. The second element of the string is the integer assigned to v first child. The remaining elements are obtained by sorting the collection of integers assigned to its remaining children. If v is labeled with Q, we proceed somewhat differently. As before, the first element of v's string is the label Q, and the second element is the integer assigned to v's first child. To determine the remainder of the string, we consider two sequences, one consisting of the integers assigned to the remaining children of v in order, the second being the reverse of the first. The remainder of v's string is whichever of these two sequences is lexicographically less.

It is a simple induction to see that two height-h vertices receive the same string or same integer if and only if the subtrees rooted at these vertices are isomorphic. This shows that the isomorphism algorithm is correct.

There are $\leq 1 + \log t$ levels in the decomposition trees for t-node trees. Therefore, the above algorithm has $O(\log t)$ stages, where $t = O(n + m)$. In each stage, the most difficult step is sorting strings. The sum of the lengths of the strings is $O(t)$ in each stage. Therefore, we can assign a processor to each symbol of each string. To compare two strings, processors associated with corresponding symbols communicate to compare their symbols. Using the powerful concurrent-write capability, the processors associated with the two strings can determine in constant time the minimum index at which the strings differ, and hence which string comes first in lexicographic order. Using a logarithmic-time comparison sort [9], the strings can then be sorted in $O(\log t)$ time using $O(t)$ processors. Hence t-node tree isomorphism can be tested in $O(\log^2 t)$ time using t processors.

Acknowledgments. Many thanks to David Shmoys, who advised the thesis based in part on this research. Thanks also to others with whom I discussed this research, including Dina Kravets, Tom Leighton, Charles Leiserson, George Lueker, Mark Novick, James Park, John Reif, Cliff Stein, and Joel Wein.

REFERENCES

[1] A. AHO, J. HOPCROFT, AND J. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison–Wesley, Reading, MA, 1974.

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

[3] R. J. ANDERSON AND G. L. MILLER, *A simple randomized parallel algorithm for list-ranking*, Inform. Process. Lett., 33 (1990), p. 33.

[4] 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.

[5] C. BERGE, *Graphs and Hypergraphs*, North–Holland, Amsterdam, 1973.

[6] 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.

[7] N. CHANDRASEKHARAN AND S. S. IYENGAR, *NC algorithms for recognizing chordal graphs and k-trees*, Technical report 86-020, Department of Computer Science, Louisiana State University, Baton Rouge, LA, (1986).

[8] C. J. COLBOURN AND K. S. BOOTH, *Linear time automorphism algorithms for trees, interval graphs, and planar graphs*, SIAM J. Comput., 10 (1981), pp. 203–225.

[9] R. COLE, *Parallel merge sort*, SIAM J. Comput., 17 (1988), pp. 770–785.

[10] 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.

[11] D. COPPERSMITH AND S. WINOGRAD, *Matrix multiplication via arithmetic progressions*, SIAM J. Comput., 11 (1982), pp. 472–492.

[12] E. DAHLHAUS AND M. KARPINSKI, *The matching problem for strongly chordal graphs is in NC*, Technical report 855-CS, Institut für Informatik, Universität Bonn, Bonn, Germany, 1986.

[13] ———, *Fast parallel computation of perfect and strongly perfect elimination schemes*, Technical report 8519-CS, Institut für Informatik, Universität Bonn, Bonn, Germany, 1987.

[14] G. A. DIRAC, *On rigid circuit graphs*, Abh. Math. Sem. Univ. Hamburg, 25 (1961), pp. 71–76.

[15] A. EDENBRANDT, *Combinatorial problems in matrix computation*, TR-85-695 (Ph.D. thesis), Department of Computer Science, Cornell University, Ithaca, NY, 1985.

[16] ———, *Chordal graph recognition is in NC*, Inform. Process. Lett., 24 (1987), pp. 239–241.

[17] D. FULKERSON AND O. GROSS, *Incidence matrices and interval graphs*, Pacific J. Math., 15 (1965), pp. 835–855.

[18] F. GAVRIL, *Algorithms for minimum coloring, maximum clique, minimum covering by cliques, and maximum independent set of a chordal graph*, SIAM J. Comput., 1 (1972), pp. 180–187.

[19] H. Gazit, *An optimal randomized parallel algorithm for finding connected components in a graph*, SIAM J. Comput., 20 (1991), pp. 1046–1067.

[20] H. Gazit and G. L. Miller, *An improved parallel algorithm that computes the BFS numbering of a directed graph*, Inform. Process. Lett., 28 (1988), pp. 61–65.

[21] J. R. Gilbert, D. J. Rose, and A. Edenbrandt, *A separator theorem for chordal graphs*, SIAM J. Algebraic Discrete Meth., 5 (1984), pp. 306–313.

[22] P. C. Gilmore and A. J. Hoffman, *A characterization of comparability graphs and of interval graphs*, Canad. J. Math., 16 (1964), pp. 539–548.

[23] M. C. Golumbic, *Algorithmic Graph Theory and Perfect Graphs*, Academic Press, New York, 1980.

[24] C.-W. Ho and R. C. T. Lee, *Efficient parallel algorithms for finding maximal cliques, clique trees, and minimum coloring on chordal graphs*, Inform. Process. Lett., 28 (1988), pp. 301–309.

[25] ———, *Counting clique trees and computing perfect elimination schemes in parallel*, Inform. Process. Lett., 31 (1989), pp. 61–68.

[26] J. Hopcroft and J. Wong, *Linear time algorithm for isomorphism of planar graphs*, in Proc. 6th Annual ACM Symposium on Theory of Computing Association for Computing Machinery, New York, 1974, pp. 172–184.

[27] P. N. Klein, *Efficient parallel algorithms for planar, chordal, and interval graphs*, TR-426 (Ph.D. thesis), Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 1988.

[28] ———, *Efficient parallel algorithms for chordal graphs*, in Proc. 29th Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1989, pp. 150–161.

[29] P. N. Klein and J. H. Reif, *An efficient parallel algorithm for planarity*, J. Comput. System Sci., 37 (1988), pp. 190–246.

[30] D. Kozen, U. Vazirani, and V. Vazirani, *NC algorithms for comparability graphs,and testing for unique perfect matching*, in Proc. 5th Symposium Foundations of Software Technology and Theoretical Computer Science, Lecture Notes in Comput. Sci. 206, Springer-Verlag, New York, 1985, pp. 496–503.

[31] R. E. Ladner and M. J. Fischer, *Parallel Prefix Computation*, J. Assoc. Comput. Mach., 27 (1980), pp. 831–838.

[32] G. S. Lueker and K. S. Booth, *A linear time algorithm for deciding interval graph isomorphism*, J. Assoc. Comput. Mach., 26 (1979), pp. 183–195.

[33] G. L. Miller and J. H. Reif, *Parallel tree contraction and its application*, in Proc. 26th Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1985, pp. 478–489.

[34] J. Naor, M. Naor, and A. A. Schäffer, *Fast parallel algorithms for chordal graphs*, SIAM J. Comput., 18 (1989), pp. 327–349.

[35] M. B. Novick, personal communication, 1987.

[36] ———, personal communication, 1988.

[37] ———, *Logarithmic time parallel algorithms for recognizing comparability and interval graphs*, Technical report TR89-1015, Department of Computer Science, Cornell University, Ithaca, NY, 1989.

[38] J. H. Reif and S. Rajasekaran, *Optimal and sublogarithmic time randomized parallel sorting algorithms*, SIAM J. Comput., 18 (1989), pp. 594–607.

[39] D. J. Rose, *Triangulated graphs and the elimination process*, J. Math. Anal. Appl., 32 (1970), pp. 597–609.

[40] D. J. Rose, R. E. Tarjan, and G. S. Lueker, *Algorithmic aspects of vertex elimination on graphs*, SIAM J. Comput., 5 (1976), pp. 266–283.

[41] J. E. Savage and M. G. Wloka, *A parallel algorithm for channel routing*, in Graph-Theoretic Concepts in Computer Science, J. van Leeuwen, ed., Lecture Notes in Comput. Sci. 344, Springer-Verlag, New York, 1988, pp. 288–301.

[42] R. Schreiber, *A new implementation of sparse Gaussian elimination*, ACM Trans. Math. Software, 8 (1982), pp. 256–276.

[43] Y. Shiloach and U. Vishkin, *An $O(\log n)$ parallel connectivity algorithm*, J. Algorithms, 3 (1982), pp. 57–67.

[44] R. E. Tarjan and U. Vishkin, *Finding biconnected components and computing tree functions in logarithmic parallel time*, SIAM J. Comput., 14 (1975), pp. 862–874.

SIAM J. COMPUT.
Vol. 25, No. 4, pp. 828–861, August 1996

THE SUBLOGARITHMIC ALTERNATING SPACE WORLD[*]

MACIEJ LIŚKIEWICZ[†] AND RÜDIGER REISCHUK[‡]

Abstract. This paper tries to fully characterize the properties and relationships of space classes defined by Turing machines (TMs) that use less than logarithmic space—be they deterministic, nondeterministic, or alternating (DTMs, NTMs, or ATMs). We provide several examples of specific languages and show that such machines are unable to accept these languages. The basic proof method is a nontrivial extension of the $1^n \mapsto 1^{n+n!}$ technique to alternating TMs.

Let llog denote the logarithmic function log iterated twice, and let $\Sigma_k Space(S)$ and $\Pi_k Space(S)$ be the complexity classes defined by S-space-bounded ATMs that alternate at most $k-1$ times and start in an existential (resp., universal) state. Our first result shows that for each $k > 1$, the sets

$$\Sigma_k Space(\text{llog}) \setminus \Pi_k Space(o(\text{log})) \quad \text{and}$$
$$\Pi_k Space(\text{llog}) \setminus \Sigma_k Space(o(\text{log}))$$

are both not empty. This implies that for each $S \in \Omega(\text{llog}) \cap o(\text{log})$, the classes

$$\Sigma_1 Space(S) \subset \Sigma_2 Space(S) \subset \Sigma_3 Space(S) \subset \cdots$$
$$\subset \Sigma_k Space(S) \subset \Sigma_{k+1} Space(S) \subset \cdots$$

form an infinite hierarchy. Furthermore, this separation is extended to space classes defined by ATMs with a nonconstant alternation bound A provided that the product $A \cdot S$ grows sublogarithmically.

These lower bounds can also be used to show that basic closure properties do not hold for such classes. We obtain that for any $S \in \Omega(\text{llog}) \cap o(\text{log})$ and all $k > 1$, $\Sigma_k Space(S)$ and $\Pi_k Space(S)$ are not closed under complementation and concatenation. Moreover, $\Sigma_k Space(S)$ is not closed under intersection and $\Pi_k Space(S)$ is not closed under union.

It is also shown that ATMs recognizing bounded languages can always be guaranteed to halt. For the class of Z-bounded languages with $Z \leq \exp S$, we obtain the equality $co\text{-}\Sigma_k Space(S) = \Pi_k Space(S)$.

Finally, for sublogarithmic bounded ATMs, we give a separation between the weak and strong space measure and prove a logarithmic lower space bound for the recognition of nonregular context-free languages.

Key words. space complexity, sublogarithmic complexity bounds, alternating Turing machines, halting computations, complementation of languages, complexity hierarchies, closure properties, context-free languages, bounded languages

AMS subject classifications. 68Q05, 68Q10, 68Q25, 68Q45

1. Introduction.

It is well known that if a deterministic or nondeterministic Turing machine (DTM or NTM, respectively) uses less than llog space, then the machine can recognize only regular languages; it is also well known that there exist nonregular languages in $DSpace(\text{llog})$. Therefore, let SUBLOG $:= \Omega(\text{llog}) \cap o(\text{log})$ denote the set of all nontrivial sublogarithmic space bounds, where llog abbreviates the twice-iterated logarithmic function $n \mapsto \lfloor \log\log n \rfloor$. On the other hand, the logarithm seems to be the most dramatic bound for space complexity since most techniques used in space complexity investigations only work for bounds above this threshold. There are several important results for such space classes known (for a general overview, see, for example, [18]), and it is an open question if they also hold for space bounds between llog and log. One of the most exciting problems of this type is whether the closure under complement for NTMs

$$NSpace(S) = co\text{-}NSpace(S),$$

[*] Received by the editors July 22, 1993; accepted for publication (in revised form) December 19, 1994. Most of this research was performed while both authors were working at the Technische Hochschule Darmstadt, Darmstadt, Germany. Some of this research was performed at the International Computer Science Institute, Berkeley, CA.

[†] Instytut Informatyki, Uniwersytet Wrocławski, Przesmyckiego 20, 51-151 Wrocław, Poland (liskiewi@ii.uni.wroc.pl). The research of this author was supported by the Alexander-von-Humboldt-Stiftung and KBN grant 211979101.

[‡] Institut für Theoretische Informatik, Medizinische Universität zu Lübeck, Wallstraße 40, 23560 Lübeck, Germany (reischuk@informatik.mu-luebeck.de).

shown by Immerman and Szelépcsenyi [13], [22], remains valid for sublogarithmic space bounds. If this equality were not valid for a function $S \in$ SUBLOG then obviously $DSpace(S) \subset NSpace(S)$.[1]

A special situation holds for bounded languages containing only strings of a certain block structure.

DEFINITION 1. *Let $Z : \mathbb{N} \to \mathbb{N}$ be a function. Then a language $L \subseteq \{0, 1\}^*$ is* Z-bounded *if each $X \in L$ contains at most $Z(|X|)$ zeros. L is* bounded *if it is Z-bounded for some constant function Z.*

Recently Alt, Geffert, and Mehlhorn [2] and, independently, Szepietowski [23] have proved that for the class of Z-bounded languages, where Z is a constant or a small growing function, the closure under complement holds, which means that in this case $NSpace(S) = co\text{-}NSpace(S)$, even for sublogarithmic bounds. Still, we conjecture that, in general, the above result does not hold. Towards this, we will prove in this paper that $\Sigma_k Space(S)$ is not closed under complementation for any $S \in$ SUBLOG and all $k > 1$.

Recall that for $k \geq 1$, the class $\Sigma_k Space(S)$ is defined as all languages that can be accepted by alternating S-space-bounded TMs making at most $k - 1$ alternations and starting in an existential state. $\Pi_k Space(S)$ denotes the set of languages accepted by the same kind of machines, except that they start in a universal state. By definition, $\Sigma_1 Space(S) = NSpace(S)$. We will also consider alternating TMs (ATMs) with a nonconstant bound A for the number of alternations. In this case, the notations $\Sigma_A Space(S)$ and $\Pi_A Space(S)$ are used.

By standard techniques, it follows from Immerman and Szelépcsenyi's theorem that for $S \in \Omega(\log)$ and for all $k \geq 1$,

$$\Sigma_1 Space(S) = \Sigma_k Space(S) = \Pi_k Space(S).$$

Note that these techniques do not work for sublogarithmic space bounds. Recently, Chang et al. [7] have shown that there is a language in $\Pi_2 Space(\text{llog})$ that does not belong to $NSpace(o(\log))$. Clearly, this proves that for space bounds S in SUBLOG, the alternating S-space hierarchy does not collapse to the first level and that

$$\Sigma_1 Space(S) \subset \Pi_2 Space(S).$$

It was left as an open problem whether the whole alternating hierarchy for sublogarithmic space is strict (see [5], [9], [10], [14]). Here we will prove that the problem has a positive answer.

We develop techniques to investigate properties of sublogarithmic computations and then generalize them to an inductive proof that the separation of the $\Sigma_k Space(S)$ and $\Pi_k Space(S)$ classes holds for all levels k. The base case is the existence of a language that separates $\Pi_2 Space(\text{llog})$ from $\Sigma_2 Space(o(\log))$. Its complement separates $\Sigma_2 Space(\text{llog})$ from $\Pi_2 Space(o(\log))$.

Inductively, we will construct a sequence of languages $L_{\Sigma k}$ and $L_{\Pi k}$ and prove that $L_{\Sigma k}$ can be recognized by a Σ_kTM with $\text{llog}\, n$ space, but not by any Π_kTM that is $o(\log)$-space bounded. The corresponding claim interchanging Σ_k and Π_k holds for $L_{\Pi k}$. For this purpose, for infinitely many n, we will explicitly pinpoint a pair of strings, one string in $L_{\Sigma k}$ and the other one in $L_{\Pi k}$, and show that any sublogarithmic space-bounded Σ_kTM or Π_kTM will make an error on at least one of these strings. Thus we obtain the following result.

THEOREM 1. *For all $k > 1$, the following hold:*

$$\Sigma_k Space(\text{llog}) \setminus \Pi_k Space(o(\log)) \neq \emptyset \quad \textit{and}$$
$$\Pi_k Space(\text{llog}) \setminus \Sigma_k Space(o(\log)) \neq \emptyset.$$

[1] In [25, p. 419], it is incorrectly cited that $DSpace(S) \subset NSpace(S)$ for $S \in$ SUBLOG. Thus the problem if $DSpace(S) = NSpace(S)$ is still open for any $S \in \Omega(\text{llog})$ (see Remark 6.1 in [17]).

This result gives a complete and best possible separation for the sublogarithmic space world, except for the first level $k = 1$. It is left open whether also $\Sigma_1 \mathit{Space}(S) \neq \Pi_1 \mathit{Space}(S)$ for $S \in$ SUBLOG. The current techniques do not seem to be applicable to this case.

This separation implies that the sublogarithmic space hierarchy is an infinite one, contrary to the case for logarithmic or larger space bounds.

COROLLARY 1. *For any $S \in$ SUBLOG and all $k \geq 1$, the following hold:*

$$\Sigma_k \mathit{Space}(S) \subset \Sigma_{k+1} \mathit{Space}(S),$$
$$\Pi_k \mathit{Space}(S) \subset \Pi_{k+1} \mathit{Space}(S).$$

The existence of this strict hierarchy has been shown independently by von Braunmühl et al. [6]. Geffert [11] has announced similar results. (For a chronology of events, see [24].)

Furthermore, we can generalize the separation to machines with an unbounded number of alternations.

DEFINITION 2. *A function $A : \mathbb{N} \to \mathbb{N}$ is* computable in space *S if there exists a DTM that for all inputs of the form 1^n writes down the binary representation of $A(n)$ on an extra output tape using no more than $S(n)$ work space. A is* approximable from below in space *S if there exists a function A' that is computable in space S with $A'(n) \leq A(n)$ for all $n \in \mathbb{N}$ and $A'(n) = A(n)$ for infinitely many $n \in \mathbb{N}$.*

The class of bounds that are approximable from below in space llog contains functions of logarithmic and double-logarithmic growth and also polynomials of such functions. The iterated logarithm $\log^*$ belongs to this class as well. In §3, we will discuss a specific example of logarithmic growth.

THEOREM 2. *For any pair of functions $S \in$ SUBLOG and A with $A > 1$ and $A \cdot S \in o(\log)$, where A is approximable from below in space S, the following hold:*

$$\Sigma_A \mathit{Space}(S) \setminus \Pi_A \mathit{Space}(S) \neq \emptyset,$$
$$\Pi_A \mathit{Space}(S) \setminus \Sigma_A \mathit{Space}(S) \neq \emptyset.$$

COROLLARY 2. *For any S and A as in Theorem 2, the following hold:*

$$\Sigma_A \mathit{Space}(S) \subset \Sigma_{A+1} \mathit{Space}(S),$$
$$\Pi_A \mathit{Space}(S) \subset \Pi_{A+1} \mathit{Space}(S).$$

Thus for space bounds $S \in \Omega(\text{llog})$ and approximable functions A, for example, one obtains the following relations:

1. $\bigcup_{k \in \mathbb{N}} \Sigma_k \mathit{Space}(S) \subset \Sigma_{\log^*} \mathit{Space}(S)$ if $S \in o(\frac{\log}{\log^*})$.
2. $\Sigma_A \mathit{Space}(S) \subset \Sigma_{A+1} \mathit{Space}(S)$ for $A, S \in O(\log^{1/2-\epsilon})$,
3. For $k \in \mathbb{N}$, let

$$\mathcal{ALSL}^k := \mathit{AAlterSpace}(\text{llog}^k, \text{llog}).$$

Then for any k, $\mathcal{ALSL}^k \subset \mathcal{ALSL}^{k+1}$ holds.

Note that for logarithmic bounds the corresponding question is still open, i.e., for any k, it is unknown whether

$$\mathit{AAlterSpace}(\log^k, \log) \subset \mathit{AAlterSpace}(\log^{k+1}, \log).$$

It is well known that for any function S the complexity class $\Sigma_1 \mathit{Space}(S)$ is closed under union and intersection (see, e.g., [25]). However, it is still an open problem whether for $S \in$ SUBLOG the class $\Sigma_1 \mathit{Space}(S)$ is closed under complementation. More generally, for arbitrary k the classes $\Sigma_k Space(S)$ are closed under union, and symmetrically the $\Pi_k Space(S)$ are closed under intersection. In [14], we have developed a technique that shows that for $S \in$ SUBLOG and $k = 2, 3$, $\Sigma_k Space(S)$ and $\Pi_k Space(S)$ are not closed under complementation. Furthermore, $\Sigma_k Space(S)$ is not closed under intersection and $\Pi_k Space(S)$ is not closed under union. Combining these ideas with the separation results above, we get the same closure properties for all levels.

THEOREM 3. *For any $S \in$ SUBLOG and all $k > 1$, $\Sigma_k Space(S)$ and $\Pi_k Space(S)$ are not closed under complementation and concatenation. Moreover, $\Sigma_k Space(S)$ is not closed under intersection and $\Pi_k Space(S)$ is not closed under union.*

Note that nonclosure under complementation for Σ_k and Π_k classes is not trivially equivalent to Theorem 1, which says that sublogarithmic $\Sigma_k Space$ and $\Pi_k Space$ are distinct. Sublogarithmic space-bounded machines do not have a counter, which could detect an infinite path of computation. It is an interesting open problem whether $\Pi_k Space(S) = co\text{-}\Sigma_k Space(S)$ for $k = 1, 2, \ldots$ (see the discussion in [14]). Here we obtain the following partial solution generalizing Sipser's result on halting space-bound computation for sublogarithimic space-bounded deterministic TMs [19]: For bounded languages it can be shown that there exist equivalent ATMs that always halt. This implies the following result.

THEOREM 4. *Let $S \in$ SUBLOG be a space bound and Z be a function computable in space S with $Z \leq \exp S$. Then for all $k \geq 1$ and every Z-bounded language $L \subseteq \{0, 1\}^*$, the following holds:*

$$L \in \Sigma_k Space(S) \iff \overline{L} \in \Pi_k Space(S).$$

Observe that for $S \geq \log$ the function Z can grow linearly and then Z does not put any restriction on the structure of the strings in L. Thus this theorem gives a smooth approximation of the fact that for at least logarithmic space bounds, Σ_k and Π_k are complementary for arbitrary languages. We conjecture that the computability of Z is needed in the claim above. Furthermore, there are some indications that the theorem might not be true generally for bounds Z much larger than $\exp S$.

Finally, we prove a logarithmic lower space bound for the recognition of context-free languages by ATMs. We will show that the deterministic context-free language $L_{\neq} := \{1^n 0 1^m \mid n \neq m\}$ does not belong to $ASpace(o(\log))$. It is interesting to note that this language—but not its complement—can be recognized even by a deterministic machine in *weak space* llog.

DEFINITION 3. *We say that an ATM M is* (strongly) S-space bounded *if on every input X it only enters configurations that use at most $S(|X|)$ space. M is* weakly S-space bounded *if for every input X that is accepted it has an accepting computation tree all of whose configurations use at most $S(|X|)$ space. $DSpace(S)$ denotes the class of languages accepted by S-space-bounded DTMs and $weakDSpace(S)$ denotes the languages accepted by weakly S-space-bounded DTMs. A corresponding notation is used for NTMs and ATMs.*

In this paper, we consider only the more natural strong requirement for space complexity. For at least logarithmic space bounds, the two conditions do not make a difference, while in the sublogarithmic case, they obviously do. When studying the closure under complement of a language L and alternating hierarchies built on it, the weak measure is not appropriate. This is because for strings in $\overline{L}$ a machine for L may use arbitrarily much space, while a machine for $\overline{L}$ would be required to be bounded. The example above shows that with respect to the weak measure already for DTM $weakDSpace(\text{llog})$ contains languages that do not belong to $co\text{-}weakDSpace(o(\log))$.

In [7], Chang et al. stated as an open problem whether weak and strong sublogarithmic space-bounded ATMs have the same power. Obviously, our lower space bound for recognizing $L_{\neq}$ by ATMs proves the following result.

THEOREM 5. $weakDSpace(\text{llog}) \setminus ASpace(o(\log)) \neq \emptyset$.

As consequences, we obtain the following corollaries.

COROLLARY 3. *For any $k \geq 1$ and each $S \in$ SUBLOG,*

$$\Sigma_k Space(S) \subset weak\Sigma_k Space(S) \quad \text{and}$$
$$\Pi_k Space(S) \subset weak\Pi_k Space(S).$$

COROLLARY 4. *For each $S \in$ SUBLOG,*

$$ASpace(S) \subset weakASpace(S).$$

We next generalize the specific lower bound above to arbitrary deterministic context-free languages, which also improves a result for NTMs shown by Alt, Mehlhorn, and Geffert [2]. Before stating the result we need the following definition (see [20] and [12]). A language L is called *strictly nonregular* if one can find strings u, v, w, x, and y such that $L \cap \{u\}\{v\}^*\{w\}\{x\}^*\{y\}$ is context free but nonregular.

THEOREM 6. *Let L be a nonregular deterministic context-free language, a strictly nonregular language, or a nonregular context-free bounded language. Then*

$$L \notin \bigcup_{k \in \mathbb{N}} \Sigma_k Space(o(\log)).$$

Furthermore, for ATMs without any bound on the number of alternations, it is not possible that L and $\overline{L}$ both belong to $ASpace(o(\log))$.

This paper is organized as follows. In the next section, the necessary technical tools for sublogarithmic space-bounded ATMs will be developed. In §3, we will define a sequence of pairs of languages indexed by the level number k to prove the sublogarithmic space hierarchy. We then investigate closure properties of sublogarithmic space classes. Section 5 is devoted to the lower space bounds for context-free languages. The paper concludes with a discussion of the most interesting open problems for sublogarithmic space classes remaining.

Preliminary versions of most of these results have been presented in [14] and [15].

2. Properties of sublogarithmic space-bounded ATMs. The TM model we consider is equipped with a two-way read-only input tape and a single read–write work tape. Moreover the input word is stored on the input tape between end markers \$.

DEFINITION 4. *A* memory state *of a TM M is an ordered triple $\alpha = (q, u, i)$, where q is a state of M, u a string over the work tape alphabet, and i a position in u (the locaton of the work-tape head). A* configuration *of M on an input X is a pair (α, j) consisting of a memory state α and a position j, with $0 \leq j \leq |X| + 1$, of the input head. $j = 0$ or $j = |X| + 1$ means that this head scans the left (resp., right) end marker. For a memory state $\alpha = (q, u, i)$, let $|\alpha|$ denote the length of the memory inscription u.*

We may assume that for a successor (α', j') of a configuration (α, j), $|\alpha'| \geq |\alpha|$ always holds. The state set of an ATM is partioned into subsets of existential, universal, accepting, and rejecting states. We say that a configuration $((q, u, i), j)$ is existential (resp., universal, accepting, or rejecting) if q has the corresponding mode. All accepting and rejecting configurations C are assumed to be terminating, i.e., there are no more configurations that can be reached from C.

DEFINITION 5. *Let*

$$(\alpha, i) \models^{\star}_{M,X} (\beta, j)$$

denote the property that the ATM M with X on its input tape has a computation path $C_1 = (\alpha, i), C_2, \ldots, C_t = (\beta, j)$.

$$(\alpha, i) \models_{M,X} (\beta, j)$$

denotes the same fact, but with the following restriction: $t \geq 2$ *and the mode of the configurations* $C_2, \dots, C_{t-1}$ *is the same as that of* C_1 *(i.e., if* C_1 *is existential, then all* C_l *for* $l = 2, \dots, t-1$ *are existential; otherwise, they are all universal).*

$$\mathsf{acc}_M^k(\alpha, i, X)$$

denotes the predicate that says that M *starting in configuration* (α, i) *with* X *on its input tape accepts (i.e., has an accepting subtree), and on each computation path of that tree, it makes at most* $k-1$ *alternations. Let*

$$Space_M(\alpha, i, X)$$

denote the maximum space used in configurations that M *can reach on input* X *starting in configuration* (α, i) *and* $Space_M(X) := Space_M(\alpha_0, 0, X)$*, where* $(\alpha_0, 0)$ *is the initial configuration of* M*. Similarly, let*

$$Alter_M(\alpha, i, X)$$

denote the maximum number of alternations that M *can make on input* X *starting in configuration* (α, i) *and* $Alter_M(X) := Alter_M(\alpha_0, 0, X)$.

2.1. Inputs of a periodic structure. In this section, some properties of TM computations for binary inputs of the form $Z_1 W W \dots W Z_2$ will be described. Let M be an ATM. Then for any integer $b \geq 0$, we define

$$\mathcal{M}_b \quad := \quad \#\{\alpha \mid \alpha \text{ is a memory state of } M \text{ with } |\alpha| \leq b\}.$$

The following two lemmas characterize "short" computations, i.e., computations restricted to substrings $WW \dots W$. The first one is a generalization of a result in [16].

LEMMA 1. *Assume that*

$$X = Z_1 W^n Z_2,$$

where Z_1, W, Z_2 *are arbitrary binary strings and* $n \in \mathbb{N}$*. Moreover, let* b *be an integer and* (α, i) *and* (β, j) *be configurations with* $|\alpha| \leq |\beta| \leq b$ *and* $|Z_1| < i, j \leq |Z_1 W^n|$*. Then the following holds:*

- *If* M *can go from* (α, i) *to* (β, j) *without any alternation and without moving the input head out of the substring* W^n*, then* M *can also do so such that the head never moves* $\mathcal{M}_b^2 \cdot |W|$ *or more positions to the left of* $\min(i, j)$ *nor to the right of* $\max(i, j)$*.*

Proof. We only sketch the main idea. Assume $i \leq j$ and denote by $i_{\min}$ and $j_{\max}$ the furthest position to the left (resp., right) of the input head in the computation path of M that starts in (α, i) and ends in (β, j). Let M go from (α, i) to (β, j), moving the input head $\mathcal{M}_b^2 \cdot |W|$ or more positions to the right of j, i.e., $j_{\max} - j \geq \mathcal{M}_b^2 \cdot |W|$. By the pigeonhole principle, there exist two positions j_1 and j_2, with $j < j_1 < j_2 < j_{\max}$, and two memory states α' and α'' such that (α', j_1) and (α', j_2) are the last configurations of the computation path from (α, i) to a configuration in which M was at the position $j_{\max}$. Similarly, (α'', j_1) and (α'', j_2) are the first configurations of the computation from the position $j_{\max}$ to (β, j). Then, removing the computation paths from (α', j_1) to (α', j_2) and from (α'', j_2) to (α'', j_1), we obtain a computation that starts in (α, i) and ends in (β, j) with the head never moving more than distance $j_{\max} - (j_2 - j_1) < j_{\max}$ to the right of position j. □

LEMMA 2. *Let* $|i - j| \geq \mathcal{M}_b^2 \, (\mathcal{M}_b + 1) \cdot |W|$*. Assume that* M *can go from configuration* (α, i) *to configuration* (β, j)

(♠) *without alternating and without leaving the region between the input positions* i *and* j*.*

Then

- *there exists an integer* $c \in [1..\mathcal{M}_b]$ *such that for all* $d \in [1..\mathcal{M}_b]$ *there is a computation path satisfying* (♠) *which starts in configuration* (α, i) *and ends in* $(\beta, j - d \cdot \operatorname{sgn}(j-i) \cdot c \cdot |W|)$, *where* $\operatorname{sgn}(z) := z/|z|$;
- *moreover, there also exists a computation path satisfying* (♠) *that starts in* $(\alpha, i + d \cdot \operatorname{sgn}(j-i) \cdot c \cdot |W|)$ *and ends in* (β, j).

Proof. In the following, we will only discuss the case $i < j$ when considering the computation from configuration (α, i) to (β, j). Let $|i - j| \geq \mathcal{M}_b^2(\mathcal{M}_b + 1) \cdot |W|$.

Define the function $h(p) := i + p \cdot |W|$ for integers $p \geq 0$ and let $t := \mathcal{M}_b^2$. Partition integers in $[1..\mathcal{M}_b^3]$ into the t intervals $[L_1, R_1], [L_2, R_2], \ldots, [L_t, R_t]$ of equal length $\mathcal{M}_b$ with boundaries

$$\begin{aligned} L_s &:= (s-1)(\mathcal{M}_b + 1) + 1, \\ R_s &:= L_s + \mathcal{M}_b. \end{aligned}$$

For $s \in [1..t]$, consider all input positions $h(p)$ with $p \in [L_s, R_s]$ and the last configuration of M (before (β, j)) that visits position $h(p)$. Among these $\mathcal{M}_b + 1$ configurations, there must exist a pair with positions $p_s < q_s \in [L_s, R_s]$ and identical memory states α_s. configuration

Let $(\gamma_1, i_1) \models_{i,j} (\gamma_2, i_2)$ denote the same property as $(\gamma_1, i_1) \models_{M,X} (\gamma_2, i_2)$, but with the restriction that M going from (γ_1, i_1) to (γ_2, i_2) does not move the head to the left of i nor to the right of j. Then we can write

$$\begin{aligned} (\alpha, i) \quad \models_{i,j} \quad & (\alpha_1, h(p_1)) \models_{i,j} (\alpha_1, h(q_1)) \quad \models_{i,j} \\ & (\alpha_2, h(p_2)) \models_{i,j} (\alpha_2, h(q_2)) \quad \models_{i,j} \\ & \qquad \ldots \\ & (\alpha_t, h(p_t)) \models_{i,j} (\alpha_t, h(q_t)) \quad \models_{i,j} \quad (\beta, j). \end{aligned}$$

Since there are t pairs (p_s, q_s) and the difference between any pair is at most $\mathcal{M}_b$, by the pigeonhole principle there exists an integer $c \in [1 \ldots \mathcal{M}_b]$ and $t/\mathcal{M}_b = \mathcal{M}_b$ pairs $(p_{s_1}, q_{s_1}), (p_{s_2}, q_{s_2}), \ldots$ with identical difference c, which means that $q_{s_\ell} - p_{s_\ell} = c$ for $\ell = 1, 2, \ldots, \mathcal{M}_b$. Define $\delta' := c \cdot |W|$.

Let d be an arbitrary integer in $[1..\mathcal{M}_b]$ and define $\alpha'_\ell := \alpha_{s_\ell}$ and $i_\ell := h(p_{s_\ell})$. Then we obtain

$$\begin{aligned} (\alpha, i) \quad \models_{i,j} \quad & (\alpha'_1, i_1) \models_{i,j} (\alpha'_1, i_1 + \delta') \quad \models_{i,j} \\ & (\alpha'_2, i_2) \models_{i,j} (\alpha'_2, i_2 + \delta') \quad \models_{i,j} \\ & \qquad \ldots \\ & (\alpha'_d, i_d) \models_{i,j} (\alpha'_d, i_d + \delta') \quad \models_{i,j} \quad (\alpha'_{d+1}, i_{d+1}) := (\beta, j). \end{aligned}$$

The input X contains a sequence of identical blocks W between the positions i and j. For any $\ell \in [1 \ldots d]$, M starting in $(\alpha'_\ell, i_\ell + \delta')$ reaches $(\alpha'_{\ell+1}, i_{\ell+1})$ without moving the head to the left of $i_\ell + \delta'$. Therefore M making the same sequence of moves reaches $(\alpha'_{\ell+1}, i_{\ell+1} - \ell\delta')$ when starting in $(\alpha'_\ell, i_\ell + (\ell - 1)\delta')$. Thus we obtain

$$\begin{aligned} (\alpha, i) \quad \models_{i,j} \quad & (\alpha'_1, i_1) \quad \models_{i,j} \\ & (\alpha'_2, i_2 - \delta') \quad \models_{i,j} \\ & \qquad \ldots \\ & (\alpha'_d, i_d - (d-1)\delta') \quad \models_{i,j} \quad (\beta, j - d\delta'), \end{aligned}$$

which proves that $(\alpha, i) \models_{i,j} (\beta, j - \delta)$ for $\delta := d \cdot c \cdot |W|$. In a similar way, one can show that there exsists a computation path that starts in configuration $(\alpha, i + \delta)$ and ends in (β, j). □

In the following, M will always denote an arbitrary ATM and S a space bound in $o(\log)$. Depending on M and S, we choose a constant $\mathcal{N}_{M,S} \geq 2^8$ such that for all $n \geq \mathcal{N}_{M,S}$

$$(\mathcal{M}^6_{S(n)} + 1)^2 \quad < \quad n$$

and

$$S(n) \quad < \quad \frac{1}{2}\log n \ - 2.$$

Remark. In this section, all claims following hold for any integer $n \geq \mathcal{N}_{M,S}$.

In [8], Geffert has shown that for sublogarithmic space-bounded computations for any natural number ℓ the behavior of a nondeterministic TM on input $1^{n+\ell n!}$ is exactly the same as on 1^n. The proof is based on the so-called "$n \mapsto n + n!$ technique" developed by Stearns, Hartmanis, and Lewis in [21]. We will show that a corresponding property holds for ATMs and for all inputs of the form

$$X \ = \ Z_1 \ W^n \ Z_2 \qquad \text{and} \qquad Y \ = \ Z_1 \ W^{n+\ell n!} \ Z_2,$$

where Z_1, Z_2, and W are arbitrary binary strings and $\ell \in \mathbb{N}$.

Since in the following we will often compare computations on such an input X and a pumped version Y, let us introduce a special notation for positions within these strings. If i is a position within X outside the pumped region W^n, that means for the example above, either in Z_1 or in Z_2; then $\hat{i}$ denotes the corresponding position within Y. Thus

$$\hat{i} := \begin{cases} i & \text{if } i \leq |Z_1|, \\ i + |Y| - |X| & \text{if } i > |Z_1 W^n|. \end{cases}$$

The main technical tools for the analysis of sublogarithmic space-bounded ATMs are stated in the following lemmas. Here, X and Y denote strings as defined above and M is an arbitrary ATM. Note that n is now not necessarily identical to the length of the input X. Actually, X will in general be much larger than n. However, by a repeated application of the following implications, we can show that any machine M still obeys a sublogarithmic bound with respect to n.

LEMMA 3 (pumping). *Let α and β be memory states with $|\alpha| \leq |\beta| \leq S(n)$, then for any $i, j \in [0 \ \ldots \ |Z_1|] \cup [|Z_1 W^n| + 1 \ \ldots \ |X| + 1]$, the following hold:*

1. $(\alpha, i) \models_{M,X} (\beta, j) \iff (\alpha, \hat{i}) \models_{M,Y} (\beta, \hat{j})$,
2. $(\alpha, i) \models^{\star}_{M,X} (\beta, j) \iff (\alpha, \hat{i}) \models^{\star}_{M,Y} (\beta, \hat{j})$.

In the analysis below we will use the pumping lemma (Lemma 3) in the following more general form.

LEMMA 4. *Let n and m be integers with $\mathcal{N}_{M,S} \leq m \leq (n+1)^2$ and let α and β be memory states with $|\alpha| \leq |\beta| \leq S(m)$. Then for any $i, j \in [0 \ \ldots \ |Z_1|] \cup [|Z_1 W^n| + 1 \ \ldots \ |X| + 1]$, properties* 1 *and* 2 *above hold.*

These claims can be proven using the method developed in [8] and the fact that $\mathcal{M}^6_{S(m)} < n$.

2.2. Space and alternation bounds.

LEMMA 5 (small space bound).

$$Space_M(X) \ \leq \ S(n) \qquad \Longrightarrow \qquad Space_M(Y) \ = \ Space_M(X).$$

Proof. Let $Space_M(X) \leq S(n)$. Assume, to the contrary, that $Space_M(Y) \neq Space_M(X)$. We will show that $Space_M(Y) > Space_M(X)$ cannot occur. A similar contradiction can be obtained for the case $Space_M(Y) < Space_M(X)$.

Assume that $Space_M(Y) > Space_M(X)$. Hence for Y, there exists a computation path C that starts in the initial configuration $(\alpha_0, 0)$ and ends in a configuration $(\alpha, \hat{j})$ with

$|\alpha| = \mathit{Space}_M(X)$ such that from $(\alpha, \hat{j})$ M can reach a configuration $(\beta, \hat{j}')$ with $|\beta| = \mathit{Space}_M(X) + 1$ in one step:

$$(\alpha_0, 0) \models^\star_{M,Y} (\alpha, \hat{j}) \models^\star_{M,Y} (\beta, \hat{j}').$$

If j fulfills the condition $j \leq |Z_1|$ or $j > |Z_1 W^n|$ of the pumping lemma, then one can conclude immediately that

$$(\alpha_0, 0) \models^\star_{M,X} (\alpha, j) \models^\star_{M,X} (\beta, j').$$

Otherwise, using a similar pumping argument, one can show that M on input X can reach a configuration $(\alpha, \bar{j})$ in which the input head is located on W^n and reads the same symbol as in $(\alpha, \hat{j})$. Thus it can also get to memory state β in one more step. We get a contradiction since $|\beta| > \mathit{Space}_M(X)$. □

LEMMA 6 (small alternation bound).

$$\mathit{Space}_M(X) \leq S(n) \ \ \mathit{and}\ \ \mathit{Alter}_M(X) \leq \exp S(n) \implies \mathit{Alter}_M(Y) = \mathit{Alter}_M(X).$$

Proof. Let i be an integer with $i \in \{|Z_1|, |Z_1 W^n| + 1\}$ and let α be a memory state with

$$\mathit{Space}_M(\alpha, i, X) \ \leq \ S(n) \qquad \text{and} \qquad \mathit{Alter}_M(\alpha, i, X) \ \leq \ \exp S(n).$$

Assume that k is an arbitrary positive integer and let

$$\delta_k := k \cdot \mathcal{M}_b^2 \cdot (\mathcal{M}_b + 1) \cdot |W|,$$

where $b := S(n)$. We first show that for the input Y, the following claim holds.

CLAIM 1. *Let M starting in $(\alpha, \hat{i})$ alternate $k - 1$ times and never move the input head beyond $W^{n+\ell n!}$. Then there exists a computation of M with $k - 1$ alternations that also starts in $(\alpha, \hat{i})$, but in which the input head is never moved farther than δ_k positions to the right of $\hat{i}$ if $\hat{i} = |Z_1|$ (resp., to the left of $\hat{i}$ if $\hat{i} = |Z_1 W^{n+\ell n!}| + 1$).*

Proof. We prove this claim for $\hat{i} = |Z_1|$. The case $\hat{i} = |Z_1 W^{n+\ell n!}| + 1$ can be treated similarly. Let us note first that for integers k such that $\delta_k \geq n + \ell n!$, the claim holds trivially. Therefore, in the proof below, we consider only k with $\delta_k < n + \ell n!$.

Let i' be the smallest integer such that M starting in $(\alpha, \hat{i})$ makes $k - 1$ alternations with the head never moving to the left of i nor to the right of i'. Assume, to the contrary, that $i' > i + \delta_k$. Therefore, by the pigeonhole principle, there is an interval $[L, R]$ with

$$i + \mathcal{M}_b^2 \cdot |W| \leq L < R \leq i' \qquad \text{and} \qquad R - L \geq \mathcal{M}_b^2 \cdot (\mathcal{M}_b + 1) \cdot |W|,$$

and a computation path $\mathcal{C}$ of $k - 1$ alternations such that M with the head position in $[L, R]$ does not alternate.

Let $\mathcal{C}'$ be a subsequence of configurations of $\mathcal{C}$ of the maximal length such that all configurations of $\mathcal{C}'$ have the head position greater or equal to L and there is a configuration in $\mathcal{C}'$ with the head position i'. Note that the first configuration of $\mathcal{C}'$ equals (α_L, L) for some memory state α_L. Moreover, there is a configuration in $\mathcal{C}'$ with the head position R. Let (α_R, R) denote the first such configuration.

Below we show how to cut and paste $\mathcal{C}'$ to obtain a computation path of the same number of alternations but with the head never reaching the position i'. This yields a contradiction to the assumption that $i' > i + \delta_k$.

Let us first consider that $\mathcal{C}'$ is a tail of $\mathcal{C}$. By Lemma 2, there exists a constant c with $1 \leq c \leq \mathcal{M}_b$ such that M starting in (α_L, L) reaches $(\alpha_R, R - c \cdot |W|)$ with the head positions in $[L, R]$. If, additionally, M starting in $(\alpha_R, R - c \cdot |W|)$ makes the same sequence of moves as in $\mathcal{C}'$ when started in (α_R, R), then we obtain a computation for M with the same number of alternations as in $\mathcal{C}'$ but with the head never moving to the right of $i' - c \cdot |W|$.

Now assume that $\mathcal{C}'$ is not a tail of $\mathcal{C}$. Then the last configuration of $\mathcal{C}'$ has a form (α'_L, L) for some memory state α'_L. Let (α'_R, R) be the last configuration in $\mathcal{C}'$ with the head position R. By Lemma 2, there exist constants c_1 and c_2 with $1 \leq c_1, c_2 \leq \mathcal{M}_b$ such that M starting in (α_L, L) reaches $(\alpha_R, R - c_1c_2 \cdot |W|)$ and starting in $(\alpha'_R, R - c_1c_2 \cdot |W|)$ reaches (α'_L, L). It is obvious that M, starting in $(\alpha_R, R - c_1c_2 \cdot |W|)$ and making the same sequence of moves as between (α_R, R) and (α'_R, R) in $\mathcal{C}'$, reaches $(\alpha'_R, R - c_1c_2 \cdot |W|)$. Hence we obtain a computation path of the same number of alternations as in $\mathcal{C}'$ that also starts and ends in (α_L, L) and (α'_L, L), respectively, but with the head never moving to the right of $i' - c_1c_2 \cdot |W|$. □

Note that by Claim 1 and the assumption that $Alter_M(\alpha, i, X) \leq \exp S(n)$, it follows that if M with Y on the input tape starts in $(\alpha, \hat{i})$ and makes $k - 1$ alternations with the head never moved beyond $W^{n+\ell n!}$, then $k - 1 \leq \exp S(n)$. To see this, assume the opposite. Then by Claim 1, M starting in $(\alpha, \hat{i})$ makes $k - 1 = \exp S(n) + 1$ alternations such that the head is never moved farther than δ_k positions from $\hat{i}$. By the assumption that $n \geq N_{M,S}$, we conclude

$$\delta_k = (2^{S(n)}+2) \cdot \mathcal{M}_b^2 \cdot (\mathcal{M}_b+1) \cdot |W| \leq 2^{\frac{1}{2}\log n} \cdot (\mathcal{M}_b^3+1) \cdot |W| \leq n^{1/2} \cdot n^{1/2} \cdot |W| = n \cdot |W|,$$

which means that M can make the same computation on X. We obtain a contradiction since $Alter_M(\alpha, i, X) \leq \exp S(n)$. Hence our lemma follows from Claim 1 and from the following claim.

CLAIM 2. *For $k - 1 \leq \exp S(n)$ and for any memory state β and any integer $j \in \{|Z_1|, |Z_1W^n| + 1\}$, the following holds:*

M starting in (α, i) with X on the input tape reaches (β, j) with $k - 1$ alternations iff M starting in $(\alpha, \hat{i})$ with the input Y reaches $(\beta, \hat{j})$ with $k - 1$ alternations.

Proof. We prove the claim for $i = |Z_1|$ and $j = |Z_1W^n| + 1$. In the other cases, a similar proof can be used.

Assume that on input X, M reaches (β, j) from (α, i), making $k - 1$ alternations. Since

$$k \leq \exp S(n) + 1 \leq \lfloor\sqrt{n}\rfloor/2,$$

there exist nonnegative integers n_1, n_2, and n_3, with

$$\text{(i)} \qquad n_1 + n_2 + n_3 = n \quad \text{and} \quad \lfloor\sqrt{n}\rfloor \leq n_2 \leq n,$$

such that M alternates only on the prefix $Z_1W^{n_1}$ and on the suffix $W^{n_3}Z_2$ but not on W^{n_2}. By Lemma 4, for

$$i', j' \in [0 \ldots |Z_1W^{n_1}|] \cup [|Z_1W^{n_1+n_2}| + 1 \ldots |X| + 1]$$

and $m' := n,\ n' := n_2$, and $\ell' := \ell n(n-1) \cdots (n_2 + 1)$,

$$(\alpha', i') \models_{M,X} (\beta', j') \quad \Longleftrightarrow \quad (\alpha', \hat{i'}) \models_{M,Y} (\beta', \hat{j'})$$

for any configurations (α', i') and (β', j') that are reachable by M on the computation path between (α, i) and (β, j). Using this property, we can easily obtain a path with $(k - 1)$ alternations for input Y that starts in $(\alpha, \hat{i})$ and ends in $(\beta, \hat{j})$.

On the other hand, if for integers n_1, n_2, and n_3 fulfilling (i) there is a computation path for M on Y which starts in $(\alpha, \hat{i})$ and ends in $(\beta, \hat{j})$ and such that M does not alternate with the head position in $[|Z_1W^{n_1}| + 1 \ldots |Z_1W^{n_1+n_2+\ell' n_2!}|]$, then, applying Lemma 4 in the same way as above, one can construct a computation path for the input X which starts and ends in (α, i)

and (β, j), respectively, and has the same number of alternations. Therefore, to complete the proof, we have to show that there exists such a computation path for Y if we assume that M started in $(\alpha, \hat{i})$ reaches $(\beta, \hat{j})$, making $k-1$ alternations.

Let m be the largest integer such that for some $n_1, n_3 \in \mathbb{N}$, with $n_1 + m + n_3 = n + \ell n!$, there is a computation path $\mathcal{C}$ between $(\alpha, \hat{i})$ and $(\beta, \hat{j})$ of $k-1$ alternations such that M alternates only on the prefix $Z_1 W^{n_1}$ and suffix $W^{n_3} Z_2$. Assume to the contrary that

$$m < \lfloor\sqrt{n}\rfloor + \ell' \lfloor\sqrt{n}\rfloor!,$$

where $\ell' := \ell n(n-1) \cdots (\lfloor\sqrt{n}\rfloor + 1)$. Then in either W^{n_1} or W^{n_3} there exists a substring of the form $W^{m'}$, with $m' \geq 2\lfloor\sqrt{n}\rfloor$, such that M does not alternate on $W^{m'}$ either. W.l.o.g. let $W^{m'}$ be a substring of W^{n_3}. Then $W^{n_3} = W^{n'_3} W^{m'} W^{n''_3}$ for some integers n'_3 and n''_3. Below it is shown that $\mathcal{C}$ can be cut and pasted such that in the new computation path obtained M does not alternate when the input head visits W^{m+1}. This yields a contradiction to the maximality of m.

Let us define the following head-position bounds:

$$\begin{aligned} L_1 &:= |Z_1 W^{n_1}|, & R_1 &:= L_1 + |W^m| + 1, \\ L_2 &:= |Z_1 W^{n_1+m+n'_3}|, & R_2 &:= L_2 + |W^{\lfloor\sqrt{n}\rfloor}| + 1. \end{aligned}$$

Note that from the assumption that $m' \geq 2\lfloor\sqrt{n}\rfloor$ it follows that

$$\text{(ii)} \qquad R_2 + |W^{\lfloor\sqrt{n}\rfloor}| \leq |Z_1 W^{n+\ell n!}|.$$

Let $\mathcal{C}'$ be a subsequence of computations of $\mathcal{C}$ which starts and ends with the head position in $\{L_1, R_2\}$. We claim that $\mathcal{C}'$ can be modified to the computation path of the same number of alternations, which starts and ends in the same configurations as $\mathcal{C}'$ and such that M does not alternate with the head positions in $[L_1, R_1 + |W|]$. Only the case when $\mathcal{C}'$ starts and ends with the head position L_1 and R_2, respectively, will be described.

Let (α_1, L_1) be the first configuration of $\mathcal{C}'$ and (β_2, R_2) the last. Moreover, let (β_1, R_1) be the first configuration in $\mathcal{C}'$ with the head position R_1 and let (α_2, L_2) be the last one with the head position L_2. Using a similar counting argument as in the proof of Lemma 2, one can show that

$$\exists\, c_1 \in [1 \ldots \mathcal{M}_b] \quad \forall\, d \in [1 \ldots \mathcal{M}_b] \qquad (\alpha_1, L_1) \models_{M,Y} (\beta_1, R_1 + c_1 d|W|).$$

Moreover, by Lemma 2, we have

$$\exists\, c_2 \in [1 \ldots \mathcal{M}_b], \quad \forall\, d \in [1 \ldots \mathcal{M}_b], \qquad (\alpha_2, L_2 + c_2 d|W|) \models_{M,Y} (\beta_2, R_2).$$

Therefore, for $\delta := c_1 c_2 |W|$, the following hold:

$$\begin{gathered} (\alpha_1, L_1) \models_{M,Y} (\beta_1, R_1 + \delta), \\ (\alpha_2, L_2 + \delta) \models_{M,Y} (\beta_2, R_2). \end{gathered}$$

By (ii), M, making the same moves as in $\mathcal{C}'$ between (β_1, R_1) and (α_2, L_2), reaches $(\alpha_2, L_2 + \delta)$ when started in $(\beta_1, R_1 + \delta)$. Hence there is a computation path that starts in (α_1, L_1) ends in (β_2, R_2) of the same number of alternations as $\mathcal{C}'$ such that M does not alternate with the head position in $[L_1, R_2 + |W|]$. This completes the proof of Claim 2 and Lemma 6. □

2.3. Fooling ATMs by pumping the input.

LEMMA 7 (1-alternation). *For any configuration* (α, i) *with*

- $i \leq |Z_1|$ *or* $i > |Z_1\ W^n|$ *and*
- $Space_M(\alpha, i, X) \leq S(n)$ *and* $Space_M(\alpha, \hat{i}, Y) \leq S(n)$ *the following hold:*

$$\mathsf{acc}^2_M(\alpha, i, X) \implies \mathsf{acc}^2_M(\alpha, \hat{i}, Y) \qquad \textit{if } (\alpha, i) \textit{ is existential, and}$$

$$\mathsf{acc}^2_M(\alpha, \hat{i}, Y) \implies \mathsf{acc}^2_M(\alpha, i, X) \qquad \textit{for universal } (\alpha, i).$$

Proof. Assume that (α, i) fulfills both conditions above. First, let this configuration be existential and let $\mathsf{acc}^2_M(\alpha, i, X)$ be satisfied. Then there exists a universal configuration (or, if M does not alternate, a final accepting configuration) (β_0, h) with $0 \leq h \leq |X| + 1$ such that

(A) $(\alpha, i) \models_{M,X} (\beta_0, h)$, and

(B) each computation path $\mathcal{C}$ on input X that starts in (β_0, h) is finite. In addition, along each such $\mathcal{C}$, M does not alternate, and the final configuration of $\mathcal{C}$ is accepting.

We divide the string X according to h into three parts. Let $n' := \lfloor n/2 \rfloor$. Define $h_1 := |Z_1\ W^{n'}|$ if $h \leq |Z_1\ W^{n'}|$ and $h_1 := |Z_1|$ otherwise. Let $h_2 := h_1 + |W^{n'}| + 1$. Now let U denote the prefix of X of length h_1, i.e., $U := Z_1\ W^{n'}$ if $h_1 = |Z_1\ W^{n'}|$ and $U := Z_1$ otherwise. Moreover, let V denote the suffix of X of length $|X| + 1 - h_2$, i.e., if $h_1 = |Z_1\ W^{n'}|$ then $V := W^{n-2n'}\ Z_2$; else $V := W^{n-n'}\ Z_2$ (note that V can be an empty word). Then, $X = U\ W^{n'}\ V$.

For such a partition of X, the head of M in memory state (β_0, h) is located on string $\$U$ if $h \leq |Z_1\ W^{n'}|$ and on string $V\$$ otherwise. Let $a := (n' + 1)(n' + 2) \cdots n$ and let $\ell' := \ell a$. We will show that M, started in (α, i) with $X' := U\ W^{n' + \ell' n'!}\ V$ on its input tape, accepts, making at most one alternation. This proves the lemma since

$$X' = U\ W^{n' + \ell' n'!}\ V = Z_1\ W^{n + \ell' n'!}\ Z_2 = Z_1\ W^{n + \ell n!}\ Z_2 = Y.$$

Since $\mathcal{N}_{M,S} \leq n \leq (n' + 1)^2$ from Lemma 4 (for $n := n'$ and $m := n$) and by property (A), it follows that

$$(\alpha, \hat{i}) \models_{M,X'} (\beta_0, \hat{h}),$$

where $\hat{h} := h$ if $h \leq |Z_1\ W^{n'}|$ and $\hat{h} := h + \ell' n'!$ otherwise. Our lemma follows from this property and from the fact that

$$\mathsf{acc}^1_M(\beta_0, \hat{h}, X')$$

holds. Below we prove that this predicate is true.

Assume, to the contrary, that $\mathsf{acc}^1_M(\beta_0, \hat{h}, X')$ does not hold. We can distinguish two cases:

(a) $(\beta_0, \hat{h}) \models_{M,X'} (\beta, t)$ for some rejecting or existential configuration (β, t), or

(b) M starting in $(\beta_0, \hat{h})$ performs an infinite universal computation on X'.

From Lemma 4, it follows that the memory state β is reachable on X, too. We get a contradiction since by condition (B) it must hold that if M reaches a nonuniversal memory state on X then it should be accepting. Therefore, case (a) cannot occur. Below we will prove that case (b) cannot occur either. More precisely, we will show that if (b) holds then there exists an infinite universal computation path for input X which starts in $(\beta_0, \hat{h})$ and also yields a contradiction to (B).

Let $\mathcal{C}$ be an infinite universal computation path for input X' that starts in $(\beta_0, \hat{h})$. From $\mathcal{C}$ we will construct an infinite computation path for input X that also starts in $(\beta_0, \hat{h})$. Let $\hat{h}_2$ denote the index of the first symbol of the string $V\$$ on the input tape with input X', i.e., let $hh_2 := h_2 + |W^{\ell' n'!}|$. Three cases have to be distinguished.

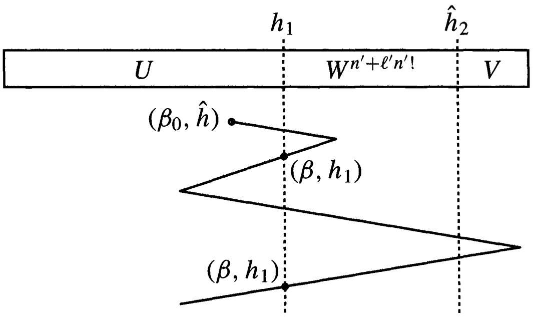


FIG. 1.

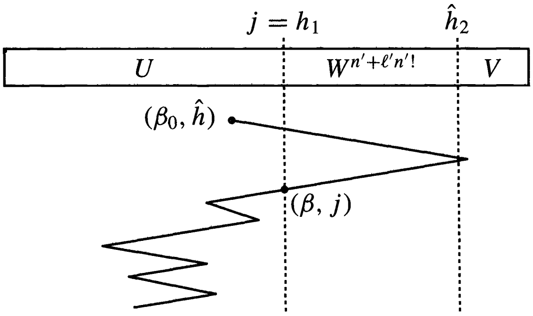


FIG. 2.

Case 1. The boundary between the prefix U and the string $W^{n'+\ell'n'!}$ or the boundary between the string $W^{n'+\ell'n'!}$ and the suffix V is crossed infinitely often in $\mathcal{C}$ (see Figure 1).

Let the boundary between the prefix U and the string $W^{n'+\ell'n'!}$ be crossed infinitely many times. Then there exists a memory state β such that the configuration (β, h_1) occurs in $\mathcal{C}$ at least twice. From Lemma 4, we conclude that

$$(\beta_0, \hat{h}) \models_{M,X} (\beta, h_1) \quad \text{and} \quad (\beta, h_1) \models_{M,X} (\beta, h_1).$$

Therefore, we obtain that M, starting in $(\beta_0, \hat{h})$, makes an infinite universal loop on X. The subcase when the boundary between the string $W^{n'+\ell'n'!}$ and the suffix V is crossed infinitely many times in $\mathcal{C}$ is similar to this case.

Case 2. There is an initial part $\mathcal{C}_1$ of $\mathcal{C}$ and an infinite rest $\mathcal{C}_2$ of $\mathcal{C}$ such that in $\mathcal{C}_2$, M scans only the input to the left of h_1 or to the right of $\hat{h}_2$ (see Figure 2).

Let (β, j) for $j = h_1$ or $j = \hat{h}_2$ be the last configuration of $\mathcal{C}_1$. From Lemma 4, we have that (β, j) is reachable from $(\beta_0, \hat{h})$ on X as well. Let $\mathcal{C}'_1$ denote a computation path from $(\beta_0, \hat{h})$ to (β, j) for input X. Then $\mathcal{C}'_1\mathcal{C}_2$ is an infinite computation path for X.

Case 3. There is an initial part $\mathcal{C}_1$ of $\mathcal{C}$ and an infinite rest $\mathcal{C}_2$ of $\mathcal{C}$ such that in $\mathcal{C}_2$, M scans only the string $W^{n'+\ell'n'!}$ (see Figure 3).

Let (β, j) for $j = h_1$ or $j = \hat{h}_2$ be the last configuration of $\mathcal{C}_1$. W.l.o.g. assume that $j = h_1$. Since $\mathcal{C}_2$ is infinite, there exists $h_1 < d < \hat{h}_2$ and a memory state γ such that (γ, d) occurs on $\mathcal{C}_2$ at least twice. By assumption, all memory states on the computation path between the two instances of (γ, d) use at most $S(n)$ space. Lemma 1 implies that there exists a computation path $\mathcal{D}$ such that $\mathcal{D}$ starts and ends in (γ, d) and the input head is never moved farther than $\mathcal{M}^2_{S(n)} \cdot |W|$ positions to the left or right of d. Let $\mathcal{C}^1_2$ denote the part of $\mathcal{C}_2$ between

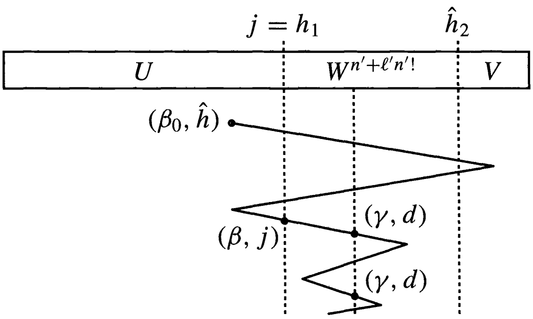


FIG. 3.

(β, j) and the first (γ, d) on $\mathcal{C}_2$. Using Lemmas 1 and 2, one can easily construct from $\mathcal{C}_2^1$ a computation path $\mathcal{D}^1$ such that

- $\mathcal{D}^1$ starts in (β, j),
- $\mathcal{D}^1$ ends in (γ, d') for some d' such that

$$d' < j + \mathcal{M}^2_{S(n)}(\mathcal{M}_{S(n)} + 1) \cdot |W| \quad \text{and} \quad d' \geq \min(d, j + \mathcal{M}^3_{S(n)} \cdot |W|),$$

and

- the input head is never moved to the left of j nor to the right of

$$j + \mathcal{M}^2_{S(n)}(\mathcal{M}_{S(n)} + 2) \cdot |W| \;\leq\; j + n' \cdot |W|.$$

Finally, let $\mathcal{C}_1'$ denote a computation path for input X starting in $(\beta_0, \hat{h})$ and ending in (β, j). By Lemma 4, such a path exists. M, starting in $(\beta_0, \hat{h})$ and making the same sequence of moves as in $\mathcal{C}_1'\mathcal{D}^1\mathcal{D}\mathcal{D}\mathcal{D}\ldots$, makes an infinite universal loop on X.

This completes the proof of the first implication of the lemma. Let us now assume that $\mathsf{acc}^2_M(\alpha, \hat{i}, Y)$ holds for a universal configuration (α, i). If $\mathsf{acc}^2_M(\alpha, i, X)$ is not true, then there exists an existential configuration (β_0, h) such that M, starting in (α, i) and working in universal states, reaches (β_0, h), and each computation $\mathcal{C}$ of M on X started in (β_0, h) is rejecting or along $\mathcal{C}$, M makes at least one alternation. Using similar methods as above, one can show that $\mathsf{acc}^2_M(\alpha, \hat{i}, Y)$ also does not hold—again, a contradiction. □

2.4. Fooling ATMs by shifting the input head. In the following two lemmas, we consider the influence of shifting the input head between identical copies of a fixed string W. For this purpose let us denote the shift distance by $\Delta := |W| \cdot n!$.

LEMMA 8 (configuration shift). *Let $X = Z_1\ W^{n+n!}\ W^s\ W^n\ Z_2$ be a binary string with $s \geq 1$ and let α and β be memory states with $|\alpha| \leq |\beta| \leq S(n)$. Then for any integer i with $i \leq |Z_1|$ or $i > |Z_1 W^{n+n!} W^s W^n|$ and any integers $j, \ell \in [|Z_1 W^{n+n!}|+1\ \ldots\ |Z_1\ W^{n+n!} W^s|]$ the following hold:*

1. $(\alpha, i) \models_{M,X} (\beta, j) \iff (\alpha, i) \models_{M,X} (\beta, j - \Delta)$,
2. $(\alpha, j) \models_{M,X} (\beta, \ell) \iff (\alpha, j - \Delta) \models_{M,X} (\beta, \ell - \Delta)$,
3. $(\alpha, j) \models_{M,X} (\beta, i) \iff (\alpha, j - \Delta) \models_{M,X} (\beta, i)$.

Proof. First note that the conditions on j and ℓ guarantee that all positions $j, \ell, j - \Delta$, and $\ell - \Delta$ considered are at least n blocks W away from the boundaries Z_1 and Z_2. Define

$$X' := Z_1\ W^n\ W^s\ W^n\ Z_2 \quad \text{and} \quad X'' := Z_1\ W^n\ W^s\ W^{n+n!}\ Z_2.$$

Set $\hat{i} := i$ if $i \leq |Z_1|$; otherwise, $\hat{i} := i - \Delta$. Using the pumping lemma twice—first for the input pair X, X' and then for X', X''—we obtain the following:

$$\begin{aligned}
1.\quad & (\alpha, i) \models_{M,X} (\beta, j) & \Longleftrightarrow\ & (\alpha, \hat{i}) \models_{M,X'} (\beta, j - \Delta)\\
& & \Longleftrightarrow\ & (\alpha, i) \models_{M,X''} (\beta, j - \Delta),\\
2.\quad & (\alpha, j) \models_{M,X} (\beta, \ell) & \Longleftrightarrow\ & (\alpha, j - \Delta) \models_{M,X'} (\beta, \ell - \Delta)\\
& & \Longleftrightarrow\ & (\alpha, j - \Delta) \models_{M,X''} (\beta, \ell - \Delta),\\
3.\quad & (\alpha, j) \models_{M,X} (\beta, i) & \Longleftrightarrow\ & (\alpha, j - \Delta) \models_{M,X'} (\beta, \hat{i})\\
& & \Longleftrightarrow\ & (\alpha, j - \Delta) \models_{M,X''} (\beta, i).
\end{aligned}$$

The claim of the lemma follows because $X'' = X$. □

In the inductive argument for the proof of Theorem 1 (Proposition 1 in §3 below), we have to guarantee a certain distance of the input head from the boundaries. For this purpose, we define

$$m_{k,n} := k \cdot (n + n!).$$

LEMMA 9 (position shift). *Let $k \geq 2$, r, s, and t be integers with $r, t \geq m_{k,n}$ and $s \geq 1$, and let Z_1, Z_2, $W \in \{0, 1\}^*$ be arbitrary strings. Then for input $X = Z_1\ W^r\ W^s\ W^t\ Z_2$ and for any configuration (α, i) fulfilling the requirements*

1. *$|Z_1\ W^r| < i \leq |Z_1\ W^r\ W^s|$ and*
2. *$Space_M(\alpha, i, X) \leq S(n)$ and $Space_M(\alpha, i - \Delta, X) \leq S(n)$,*

the following hold:

$$\mathsf{acc}_M^{k-1}(\alpha, i, X) \qquad \Longleftrightarrow \qquad \mathsf{acc}_M^{k-1}(\alpha, i - \Delta, X).$$

Proof. Let input X and configuration (α, i) be as above. We will only give a proof for

$$\mathsf{acc}_M^{k-1}(\alpha, i, X) \qquad \Longrightarrow \qquad \mathsf{acc}_M^{k-1}(\alpha, i - \Delta, X).$$

A similar argument yields the opposite implication. Let

$$\text{(i)} \qquad \mathsf{acc}_M^{k-1}(\alpha, i, X)$$

be true. First, we will show the following property for computations that start in $(\alpha, i - \Delta)$. Call a computation path of finite or infinite length *universal* if all its configurations are universal.

CLAIM 3. *For a universal configuration (α, i) of M on X, any universal computation path that starts in $(\alpha, i - \Delta)$ is finite.*

Proof. Let us assume, to the contrary, that there exists an infinite universal computation path that starts in $(\alpha, i - \Delta)$. Hence there exists a universal configuration (β, j) such that

$$(\alpha, i - \Delta) \models_{M,X} (\beta, j) \quad \text{and} \quad (\beta, j) \models_{M,X} (\beta, j).$$

If $|Z_1 W^n| < j \leq |Z_1 W^{r+s+t-(n+n!)}|$, then Lemma 8(2) implies

$$(\alpha, i) \models_{M,X} (\beta, j + \Delta) \quad \text{and} \quad (\beta, j + \Delta) \models_{M,X} (\beta, j + \Delta).$$

This means that in (α, i), M starts an infinite universal computation path with X on its input tape. This yields a contradiction to $\mathsf{acc}_M^{k-1}(\alpha, i, X)$. On the other hand, if $j \leq |Z_1 W^n|$ or $j > |Z_1 W^{r+s+t-(n+n!)}|$, then by Lemma 8(3),

$$(\alpha, i) \models_{M,X} (\beta, j).$$

Since $(\beta, j) \models_{M,X} (\beta, j)$, M also generates an infinite universal computation from (α, i). Note that we can apply the configuration-shift lemma (Lemma 8) to both α and β because by the second assumption $|\alpha| \leq |\beta| \leq S(n)$ holds. This ends the proof of Claim 3. □

First, we will solve the base case $k = 2$ and consider an existential configuration (α, i). Because of $\mathsf{acc}^1_M(\alpha, i, X)$, there exists an accepting (β, j) with

$$(\alpha, i) \models_{M,X} (\beta, j).$$

Using the configuration-shift lemma, we conclude that

$$\begin{array}{ll} (\alpha, i - \Delta) \models_{M,X} (\beta, j - \Delta) & \text{if } |Z_1 W^{n+n!}| < j \leq |Z_1 W^{r+s+t-n}|, \text{ and} \\ (\alpha, i - \Delta) \models_{M,X} (\beta, j) & \text{otherwise.} \end{array}$$

Since β is accepting, $\mathsf{acc}^1_M(\alpha, i - \Delta, X)$ holds.

For universal configurations (α, i), it will be shown that any terminating configuration (β, j) with $(\alpha, i - \Delta) \models_{M,X} (\beta, j)$ is accepting. Together with Claim 3, this proves that $\mathsf{acc}^1_M(\alpha, i - \Delta, X)$ holds. Let (β, j) with $(\alpha, i - \Delta) \models_{M,X} (\beta, j)$ be a final configuration. By Lemma 8,

$$(\alpha, i) \models_{M,X} (\beta, j + \Delta)$$

if $|Z_1 W^n| < j \leq |Z_1 W^{r+s+t-(n+n!)}|$; otherwise

$$(\alpha, i) \models_{M,X} (\beta, j).$$

Hence, if β is nonaccepting, then $\mathsf{acc}^1_M(\alpha, i, X)$ does not hold—a contradiction.

Now let $k > 2$ and consider existential configurations (α, i). Since, by assumption, M, starting in (α, i) with X on the input tape, accepts, there exists an existential computation path ending in a universal configuration (β, j), with

(ii) $$(\alpha, i) \models_{M,X} (\beta, j),$$

and

(iii) $$\mathsf{acc}^{k-2}_M(\beta, j, X).$$

(The trivial case that M accepts without alternations could be handled as above.) Let us divide the input $X = Z_1\ W^r\ W^s\ W^t\ Z_2$ into three regions A, B, and C as follows:

$$\begin{aligned} A &:= Z_1 W^{r-(n+n!)}, \\ B &:= W^{n!} W^n W^s W^n, \\ C &:= W^{t-n} Z_2. \end{aligned}$$

According to j, the input head position in configuration (β, j), the following situations will be distinguished.

Case 1. The input head is located in region A or C (see Figure 4(a)), i.e., $j \leq |A|$ or $j > |AB|$.

From property (ii) and Lemma 8(3), for $Z_1 := A$ and $Z_2 := C$, we obtain that

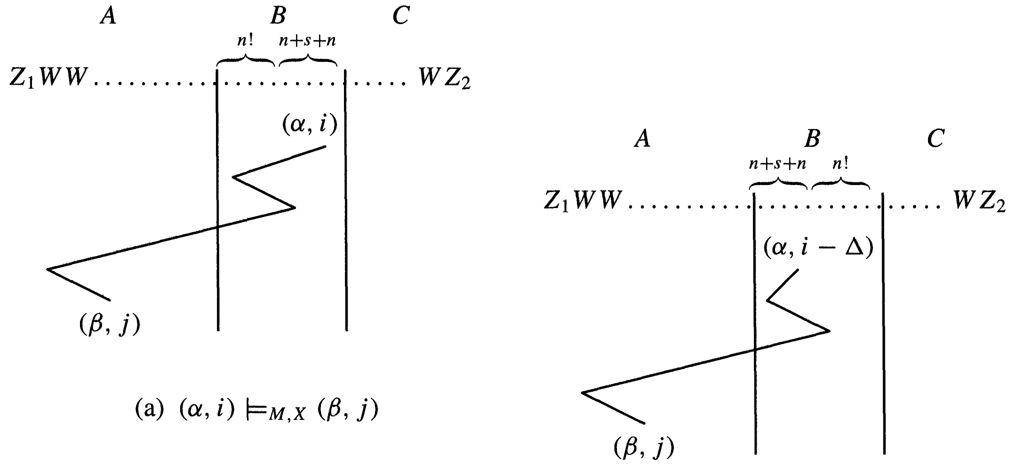
$$(\alpha, i - \Delta) \models_{M,X} (\beta, j)$$

(see Figure 4(b)). Therefore, condition (iii) implies $\mathsf{acc}^{k-1}_M(\alpha, i - \Delta, X)$.

Case 2. The input head in (β, j) visits region B (see Figure 5), i.e., $|A| < j \leq |AB|$.

In this case, using property (ii) and Lemma 8(2), for $Z'_1 := Z_1\ W^{r-2(n+n!)}$, $Z'_2 := W^{t-2n} Z_2$, and $s' := n! + n + s + n$, we conclude that

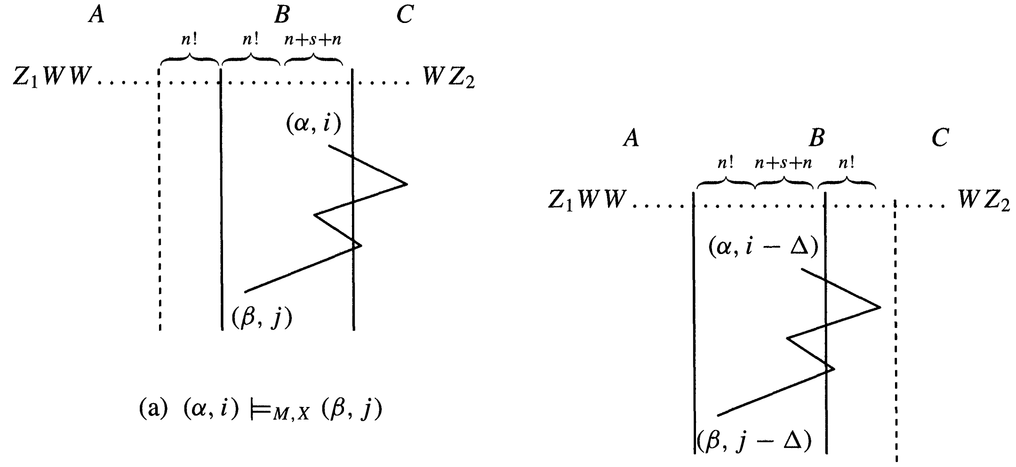
$$(\alpha, i - \Delta) \models_{M,X} (\beta, j - \Delta).$$



(a) $(\alpha, i) \models_{M,X} (\beta, j)$

(b) $(\alpha, i - \Delta) \models_{M,X} (\beta, j)$

FIG. 4.



(a) $(\alpha, i) \models_{M,X} (\beta, j)$

(b) $(\alpha, i - \Delta) \models_{M,X} (\beta, j - \Delta)$

FIG. 5.

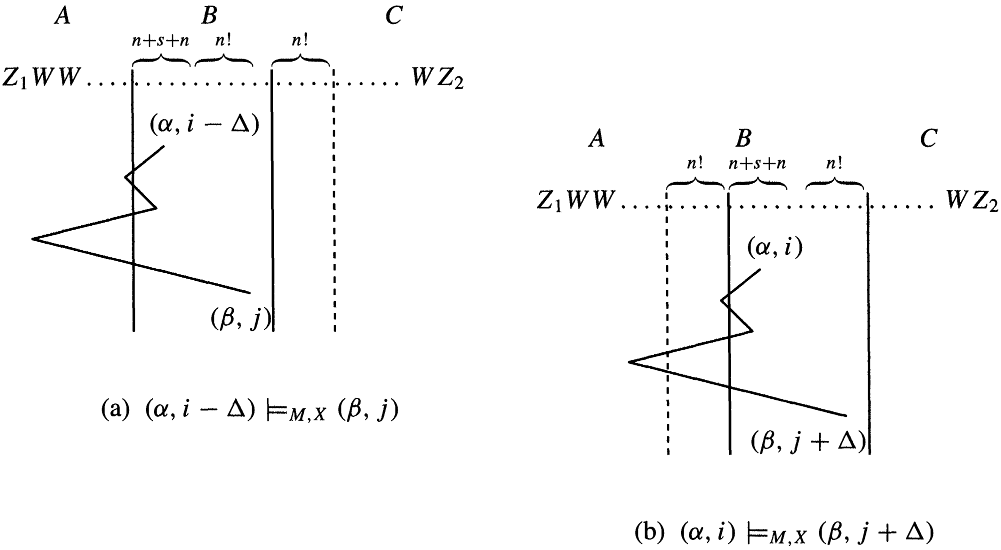
Now apply the induction hypothesis for $k-1$ with parameters $r' := r - (n + n!)$, s' and apply $t' := t - n$ to configuration (β, j). By definition of the parameters $m_{k,n}$, requirements 1 and 2 are fulfilled. Therefore, (iii) implies

$$\mathsf{acc}_M^{k-2}(\beta, j - \Delta, X),$$

and hence $\mathsf{acc}_M^{k-1}(\alpha, i - \Delta, X)$. This completes the proof for existential configurations.

For a universal (α, i), similarly to the case $k = 2$, it will be shown that for any final or existential configuration (β, j) that ends a universal computation path,

$$(\alpha, i - \Delta) \models_{M,X} (\beta, j) \quad \text{implies} \quad \mathsf{acc}_M^{k-2}(\beta, j, X).$$



(a) $(\alpha, i - \Delta) \models_{M,X} (\beta, j)$

(b) $(\alpha, i) \models_{M,X} (\beta, j + \Delta)$

FIG. 6.

Remember that, because of Claim 3, only finite paths have to be considered. Let (β, j) be such a configuration. Divide the input X into three regions A, B, and C as above. Depending on which region is visited by the input head in configuration (β, j), two cases are considered. If the input head is in region A or C (as in Figure 4(b)), then from Lemma 8(3) we obtain that $(\alpha, i) \models_{M,X} (\beta, j)$. $\mathsf{acc}_M^{k-1}(\alpha, i, X)$ thus implies $\mathsf{acc}_M^{k-2}(\beta, j, X)$.

Otherwise, the input head is located in B, i.e., $|A| < j \leq |AB|$ (see Figure 6(a)). By Lemma 8(2), one can deduce that $(\alpha, i) \models_{M,X} (\beta, j+\Delta)$, which implies $\mathsf{acc}_M^{k-2}(\beta, j+\Delta, X)$. Using the induction hypothesis for configuration $(\beta, j + \Delta)$ and for $k - 1$ with $r' := r - n$, $s' := n + s + n + n!$, and $t' := t - (n + n!)$, we obtain $\mathsf{acc}_M^{k-2}(\beta, j, X)$, which completes the proof. □

2.5. Halting computations for ATMs. Let S and Z be functions such that Z is computable in space S and $Z \leq \exp S$. We say that a binary string X is *Z-bounded* if it contains at most $Z(|X|)$ zeros.

LEMMA 10. *For every S-space-bounded ATM M, there exists an ATM M', which is also S-space-bounded, such that for all Z-bounded strings X, the following hold:*

- *M' accepts X iff M accepts X;*
- *$Alter_{M'}(X) \leq Alter_M(X)$;*
- *if $Alter_M(X) < \infty$, then every computation path of M' on X is finite.*

Proof. Let M be an ATM and let X be a Z-bounded input. In the proof below, $\mathcal{M}_b$ denotes the number of memory states of M as defined in §2.1.

Let a *crossing* be any transition of M from a configuration in which it reads an input symbol a to a configuration in which it reads an input symbol $b \neq a$, where $a, b \in \{0, 1\} \cup \{\$\}$. A sequence $\mathcal{C} = C_u, C_{u+1}, \ldots, C_v$ of consecutive configurations of a computation path on X is a *long turn* if $\mathcal{C}$ contains neither alternations nor crossings, if in C_u and C_v the input head is at the same position i for some $1 \leq i \leq |X|$, and if within $\mathcal{C}$

- either the input head visits position $i + \mathcal{M}_b^2$ but never moves to the left of i
- or it visits position $i - \mathcal{M}_b^2$, but never moves to the right of i,

where b is the amount of space used in C_v.

On the other hand, a sequence $\mathcal{C}$ without alternations or crossings is a *long hop* if the positions i and j of the input head in C_u (resp., C_v) are at least at a distance $\mathcal{M}_b^2 + 1$ apart and within $\mathcal{C}$ the input head never leaves the region between these two positions.

Now we are ready to describe the behavior of the machine M'. It first computes the value $Z(|X|)$, which by assumption can be done in space $S(|X|)$, and then simulates M step by step. Let b_t be the amount of work space used by M by its tth step.

After simulating step t of M, the machine M' stops and *rejects* iff

(a1) M rejects at this step, or

(a2) M has just finished a long turn that contains only existential configurations, or

(a3) since its last alternation, M has executed $2(Z(|X|)+1) \cdot \mathcal{M}_{b_t} + 1$ many crossings, or

(a4) within the last $2\mathcal{M}_{b_t}^3 + 1$, steps M has not made any *progress*, i.e., performed an alternation, a crossing, a long turn, or a long hop.

M' stops and *accepts* iff

(b1) M accepts, or

(b2) M has just finished a long turn that contains only universal configurations.

To check these conditions, one counter for the number of crossings, one counter for the number of steps since the last progress and a sliding window for the most recent farthest distance to the right or left, which can also be realized by counters, suffice. The length of all counters is bounded by $O(S(|X|))$. Thus, M' is $O(S)$-space bounded.

It is obvious that $Alter_{M'}(X) \leq Alter_M(X)$. To see that all computations of M' are finite, first notice that if M does not make progress inifinitely often, M' will stop the simulation eventually. Assume that M' does not stop on some path. If $Alter_M(X) < \infty$, this cannot be due to alternations or crossings of M since there is also a finite bound set by M'. Thus it remains the case that M within one block of identical input symbols performs infinitely many steps without an alternation. M' would stop if M makes a long turn; thus M has to make an unbounded number of long hops. After a long hop to one side, it cannot make a long hop to the other side because this would result in a long turn. Thus M eventually has to reach the boundary of this block and performs a crossing, a contradiction.

From Lemma 1, it follows that M' accepts the same set of Z-bounded strings as M. In case (a2), there is a shorter turn that brings M into a configuration identical to C_v. Thus if M has an accepting subtree for configuration C_u, then it still has it after the chopping of that C_v which is reached by the long turn. The dual argument holds in case (b2). Observe that in case (a3) M must have gone through a loop, and one can stop the simulation. This is because there are at most $2(Z(|X|)+1)$ different positions on the input tape (counting both directions) for performing a crossing on a Z-bounded string X. Hence at some position a memory state must repeat. A similar argument holds in case (a4) for the at most $\mathcal{M}_b^2$ many input positions that can be visited without performing a long turn or hop. □

Using this lemma, we can show the following theorem that extends Sipser's space-bounded halting result to alternating TMs.

THEOREM 7. *Let S, A, and Z be bounds with $A < \infty$ and $Z \leq \exp S$ computable in space S. Then for every S-space-bounded $\Sigma_A TM$ M, there exists a $\Sigma_A TM$ M' of space complexity S such that for all inputs X,*

- *M' accepts X iff M accepts X and X is Z-bounded, and*
- *every computation path of M' on X is finite.*

The identity of Σ_k and co-Π_k for Z-bounded languages (Theorem 4) now follows easily.

3. Hierarchies.

In this section, the separation results will be proved.

3.1. Technical preliminaries.

As a specific example of a function that can be computed in sublogarithmic space, consider the following function from [3]:

$$F(n) := \min\{k \in \mathbb{N} \mid k \text{ does not divide } n\}.$$

It is easy to see that $F \in O(\log)$. Thus on input 1^n, a TM can simply try all candidates $k = 2, 3, \ldots$ by counting the input length mod$k$ until the first nondivisor is found. Using the binary representation, this requires at most $\log F(n) \leq \operatorname{llog} n + O(1)$ space.

Obviously, F takes constant values like 2 or 3 infinitely often. We want to show that the logarithmic upper bound is also achieved infinitely often. This would imply that there exists another function G of logarithmic growth that can be approximated from below in space llog. Let $p_1 < p_2 < \cdots$ be the standard enumeration of primes and define

$$\begin{aligned}\Phi(k) &:= \prod_{p_i \leq k} p_i^{\lfloor \log_{p_i} k \rfloor},\\ \Phi^{-1}(n) &:= \min\{k \mid \Phi(k) \geq n\},\\ G(n) &:= \min\{\ell \mid \ell > \Phi^{-1}(n) \text{ and } \ell \text{ is a prime power}\}.\end{aligned}$$

The following properties can easily be derived.

1. $\Phi^{-1}(\Phi(k)) = k$ and $\Phi(\Phi^{-1}(n)) \geq n$.
2. $F(\Phi(k)) = G(\Phi(k))$, since any $\ell \leq k$ divides $\Phi(k)$ and the first nondivisor in the sequence $k+1, k+2, \ldots$ must be a prime power.
3. $F(n) \leq G(n)$ for all n, which can be seen as follows: Let $k = \Phi^{-1}(n)$. Since we have already considered the case $n = \Phi(k)$ due to property 1, we may assume $n < \Phi(k)$. By definition of Φ, there must exist a prime power $p_i^{\lfloor \log_{p_i} k \rfloor}$ that is not a divisor of n. Thus $F(n) \leq k < G(n)$.
4. $\Phi(k) = e^{k(1+o(1))}$: The prime number theorem implies
$$\prod_{p_i \leq k} p_i = e^{k(1+o(1))}.$$
Thus $\Phi(k) \geq e^{k(1+o(1))}$. On the other hand,
$$\Phi(k) \leq \prod_{p_i \leq \sqrt{k}} p_i^{\log_{p_i} k} \cdot \prod_{\sqrt{k} < p_i \leq k} p_i \leq \prod_{p_i \leq \sqrt{k}} k \cdot \prod_{p_i \leq k} p_i \leq e^{(\sqrt{k}\ \ln k + k)(1+o(1))}.$$
Hence $\Phi(k) \leq e^{k(1+o(1))}$.
5. $\Phi^{-1}(n) = \ln n\ (1 + o(1))$.
6. $G(n) = \Phi^{-1}(n)(1 + o(1)) = \ln n\ (1 + o(1))$, since any interval of the form $[\Phi^{-1}(n), \Phi^{-1}(n) \cdot (1 + o(1))]$ is guaranteed to contain a prime.

Hence, the function G is of logarithmic growth and approximated from below by F.

Let $\mathcal{F}$ be an infinite subset of the natural numbers with the following property:

$$(\clubsuit) \qquad n \in \mathcal{F} \quad \Longrightarrow \quad n + n! \notin \mathcal{F}.$$

Using the function F, we can give a simple example for such a set $\mathcal{F}$ (compare [7]):

$$\mathcal{F} := \{n > 2 \mid \forall\, \ell \in [3 \ldots n-1] \quad F(\ell) < F(n)\}.$$

The following property of $\mathcal{F}$ will be needed in the lower bound proofs.

LEMMA 11.

(1) *Every interval of the form* $[m, m^3]$ *with* $m \geq 3$ *contains an element of* $\mathcal{F}$.

(2) *For any integer* $n > 2$ *holds* $n + n! \notin \mathcal{F}$.

Proof. Since the function F is not bounded, the set $\mathcal{F}$ is infinite. More specifically, $\mathcal{F}$ contains all numbers of the form $\Phi(p_k)$ because $F(\Phi(p_k)) > p_k$ and, for all $n < \Phi(p_k)$, by the same argument as in property 3 above, $F(\Phi(p_k))) \leq p_k$. (1) can be shown by estimating the density of the sequence $(\Phi(p_k))_{k=1,2,\ldots}$. Since $p_{k+1} \leq 2p_k$ for all k, we get

$$\Phi(p_{k+1}) = \prod_{p_i \leq p_{k+1}} p_i^{\lfloor \log_{p_i} p_{k+1} \rfloor} \leq \prod_{p_i \leq p_{k+1}} p_i^{1+\lfloor \log_{p_i} p_k \rfloor} = \Phi(p_k) \cdot \prod_{p_i \leq p_{k+1}} p_i \leq \Phi(p_k)^3.$$

(2) follows easily from the equation $F(n) = F(n + n!)$. To see this equality, note that any divisor of n divides $n + n!$ as well. Hence $F(n) \leq F(n + n!)$. On the other hand, from the definition of F, we know that

$$F(n) \text{ does not divide } n$$

and, since $F(n) \leq n$,

$$F(n) \text{ divides } n!.$$

Therefore, $F(n)$ does not divide $n + n!$, which means that $F(n + n!) \leq F(n)$. □

3.2. ATMs with a constant number of alternations. With the help of the sets $\mathcal{F}$ as defined above, we construct a sequence of languages that separate the different levels of the alternation hierarchy for sublogarithmic space-bounded ATMs.

DEFINITION 6. *For an infinite subset $\mathcal{F}$ of the natural numbers, let $L_{\mathcal{F}}$ be the language over the single-letter alphabet $\{1\}$ given by $1^n \in L_{\mathcal{F}}$ iff $n \in \mathcal{F}$. Assume that $\mathcal{F}$ has property (♣) and that $L_{\mathcal{F}} \in \Pi_2 Space(\text{llog})$ and $\overline{L}_{\mathcal{F}} \in \Sigma_2 Space(\text{llog})$. Then we define $L_2 := \{1\}^+$ and, for $k \geq 3$, $L_k := (L_{k-1}\ \{0\})^+$. Furthermore,*

$$L_{\Pi 2} := L_{\mathcal{F}} \quad \textit{and} \quad L_{\Sigma 2} := \{1\}^+ \cap \overline{L}_{\mathcal{F}},$$
$$L_{\Sigma k} := \{w_1 0 w_2 0 \ldots 0 w_p 0 \mid p \in \mathbb{N},\ w_i \in L_{k-1} \textit{ and } \exists\, i \in [1 \ldots p] \quad w_i \in L_{\Pi k-1}\},$$
$$L_{\Pi k} := \{w_1 0 w_2 0 \ldots 0 w_p 0 \mid p \in \mathbb{N},\ w_i \in L_{k-1} \textit{ and } \forall\, i \in [1 \ldots p] \quad w_i \in L_{\Sigma k-1}\}.$$

Note that $L_{\Sigma 2}$ and $L_{\Pi 2}$ are just complementary. For larger k, the corresponding languages are "almost" complementary, i.e., restricting to strings with a syntactically correct division into subwords by the 0-blocks (more formally, $L_{\Pi k} = L_k \cap \overline{L}_{\Sigma k}$).

LEMMA 12. *For the specific $\mathcal{F}$ defined above, with the help of the function F, the following holds:*

$$L_{\mathcal{F}} \in \Pi_2 Space(\text{llog}) \quad \textit{and} \quad \overline{L}_{\mathcal{F}} \in \Sigma_2 Space(\text{llog}).$$

Proof. We describe llog space-bounded Π_2TMs M_Π and Σ_2TMs M_Σ that recognize the language $L_{\mathcal{F}}$ (resp., the complement of $L_{\mathcal{F}}$). The machine M_Π verifies the condition that $\forall\, \ell \in [3 \ldots n-1]$, $F(\ell) < F(n)$ as follows:

- deterministically, it computes $F(n)$ and writes down the binary representation of $F(n)$ on the tape;
- universally, it guesses an integer $\ell \in [3 \ldots n-1]$: it moves its input head to the right and stops ℓ positions from the right end of the string 1^n;
- existentially, it guesses an integer $k \in [1, \ldots, F(n) - 1]$ and then, moving the input head to the right, checks deterministically whether k divides ℓ. M_Π accepts if k does not divide ℓ.

The complementary machine M_Σ writes down $F(n)$ in binary on the work tape and tests whether

$$\exists\, \ell \in [3 \ldots n-1] \quad \forall\, k \in [1 \ldots F(n) - 1] \quad k \text{ divides } \ell.$$

Similarly as in M_Π, the input-head position represents the integer ℓ. The integer k is stored in binary on the work tape. It is obvious that M_Π recognizes $L_{\mathcal{F}}$ and that M_Σ recognizes $\overline{L}_{\mathcal{F}}$ in space $O(\text{llog})$. □

Thus the languages $L_{\mathcal{F}}$ as assumed in Definition 6 exist. For the base case of the following inductive separation, we also need the property that $L_{\mathcal{F}} \notin \Sigma_2 Space(o(\log))$ and symmetrically that $\overline{L}_{\mathcal{F}} \notin \Pi_2 Space(o(\log))$. For the example above, this has been shown explicitly in [14]. Below we give a general argument showing that this property follows simply from the condition $n \in \mathcal{F}$ and $n + n! \notin \mathcal{F}$.

LEMMA 13. *For any $k \geq 2$, the following hold:*

$$L_{\Sigma k} \in \Sigma_k Space(\text{llog}),$$
$$L_{\Pi k} \in \Pi_k Space(\text{llog}).$$

Using the fact that $L_{\mathcal{F}} \in \Pi_2 Space(\text{llog})$ and $\overline{L}_{\mathcal{F}} \in \Sigma_2 Space(\text{llog})$, the proof of these properties is simple. The separation now follows from the following result.

THEOREM 8. *For any $k \geq 2$, the following hold:*

$$L_{\Sigma k} \notin \Pi_k Space(o(\log)),$$
$$L_{\Pi k} \notin \Sigma_k Space(o(\log)).$$

We will define specific inputs that belong to $L_{\Sigma k}$ and $L_{\Pi k}$ and show that any sublogarithmic space-bounded machine cannot work correctly on both inputs.

Let $L = L_{\mathcal{F}}$ be fixed. Recall that infinitely many $n \in \mathbb{N}$ exist with $n \in \mathcal{F}$, $1^n \in L$ and $1^{n+n!} \notin L$.

DEFINITION 7. *For $n \in \mathcal{F}$, define the words*

$$W^n_{\Sigma 2} := 1^{n+n!} \quad \textit{and} \quad W^n_{\Pi 2} := 1^n,$$

and, for $k \geq 3$,

$$W^n_{\Sigma k} := \left[W^n_{\Sigma k-1}\ 0\right]^{m_{k,n}} W^n_{\Pi k-1}\ 0 \left[W^n_{\Sigma k-1}\ 0\right]^{m_{k,n}},$$

$$W^n_{\Pi k} := \left[W^n_{\Sigma k-1}\ 0\right]^{m_{k,n}} W^n_{\Sigma k-1}\ 0 \left[W^n_{\Sigma k-1}\ 0\right]^{m_{k,n}},$$

where $m_{k,n}$ are the parameters already used in the position-shift lemma (Lemma 9).

From Definition 7, the next result easily follows.

LEMMA 14. *For $k \geq 2$ and every $n \in \mathcal{F}$,*

$$W^n_{\Sigma k} \in L_{\Sigma k} \text{ and } W^n_{\Sigma k} \notin L_{\Pi k},$$
$$W^n_{\Pi k} \in L_{\Pi k} \text{ and } W^n_{\Pi k} \notin L_{\Sigma k}.$$

Let $k \geq 2$ and $S \in$ SUBLOG be a space bound. We will prove Theorem 8 by showing that if a Σ_kTM M accepts $L_{\Pi k}$ in space S, then for sufficiently large $n \in \mathcal{F}$, M accepts $W^n_{\Sigma k}$ as well. Similarly, if a Π_kTM M accepts $L_{\Sigma k}$ in space S, then for large $n \in \mathcal{F}$, it accepts $W^n_{\Pi k}$ and hence makes a mistake. Recall that $\mathcal{N}_{M,S}$ denotes the constant defined for M and S in §2.

PROPOSITION 1. *Let $S \in o(\log)$ and M be an ATM. Then for any $k \geq 2$, all $n \geq \mathcal{N}_{M,S}$, all strings $U, V \in \{0,1\}^*$, and any configuration (α, i) with*

1. *$i \leq |U|$ or $i > |U\ W^n_{\Pi k}|$ and*
2. *$Space_M(\alpha, i, UW^n_{\Pi k}V) \leq S(n)$ and $Space_M(\alpha, \hat{i}, UW^n_{\Sigma k}V) \leq S(n)$,*

the following hold:

$$\text{acc}^k_M(\alpha, i, U\ W^n_{\Pi k}\ V) \Longrightarrow \text{acc}^k_M(\alpha, \hat{i}, U\ W^n_{\Sigma k}\ V) \quad \textit{if } (\alpha, i) \textit{ is existential,}$$
$$\text{acc}^k_M(\alpha, \hat{i}, U\ W^n_{\Sigma k}\ V) \Longrightarrow \text{acc}^k_M(\alpha, i, U\ W^n_{\Pi k}\ V) \quad \textit{if } (\alpha, i) \textit{ is universal.}$$

Proof. Remember that $\hat{i}$ was defined as

$$\hat{i} = \begin{cases} i & \text{if } i \leq |U|, \\ i + (|W^n_{\Sigma k}| - |W^n_{\Pi k}|) & \text{if } i > |U\ W^n_{\Pi k}|. \end{cases}$$

For $k = 2$, the implications above follow from the 1-alternation lemma (Lemma 7).

To establish the proposition for $k > 2$, we consider the first time that the machine M makes an alternation and inductively use the corresponding properties for the strings $W^n_{\Sigma k-1}$ and $W^n_{\Pi k-1}$. The argument concentrates only on the block in the middle of a $W^n_{\Sigma k}$ string, which is a $W^n_{\Pi k-1}$ word, and analogously for $W^n_{\Pi k}$ strings with a $W^n_{\Sigma k-1}$ word in the middle. The main technical difficulty for the following argument is the possibility that in an accepting computation, the machine may just make its first alternation in the middle block and therefore may notice the difference between the $W^n_{\Sigma k}$ and $W^n_{\Pi k}$ strings. But the configuration- and position-shift lemmas (Lemmas 8 and 9, respectively) imply that there also exist accepting computations with the first alternation outside this critical region.

The details are as follows. Assume that the configuration (α, i) fulfills properties 1 and 2. Let $n \geq \mathcal{N}_{M,S}$ and define

$$\begin{aligned} X &:= U\ W^n_{\Pi k}\ V \ = \ U'\ W^n_{\Sigma k-1}\ V', \\ Y &:= U\ W^n_{\Sigma k}\ V \ = \ U'\ W^n_{\Pi k-1}\ V', \text{ where} \\ U' &:= U \left[W^n_{\Sigma k-1}\ 0 \right]^{m_{k,n}} \quad \text{and} \\ V' &:= 0 \left[W^n_{\Sigma k-1}\ 0 \right]^{m_{k,n}}\ V, \\ \Delta &:= |W^n_{\Sigma k-1}\ 0| \cdot n!, \\ \tilde{j} &:= \begin{cases} j & \text{if } j \leq |U'|, \\ j + (|W^n_{\Pi k-1}| - |W^n_{\Sigma k-1}|) & \text{if } j > |X| - |V'|. \end{cases} \end{aligned}$$

Note that $\hat{i}$ is defined with respect to the partition of the inputs X, and Y with the prefix U and suffix V, where $\tilde{j}$ is taken with respect to the prefix U' and suffix V'. Since

$$|W^n_{\Pi k-1}| - |W^n_{\Sigma k-1}| = |W^n_{\Sigma k}| - |W^n_{\Pi k}|,$$

(i) $$\hat{i} = \tilde{i} \qquad \text{whenever both values are defined.}$$

First we prove the following claim.

CLAIM 4. *For any memory state* $|\alpha_1| \leq |\alpha_2| \leq S(n)$ *and all* j_1, j_2 *with*

$$j_1, j_2 \in [0 \ldots |U'|] \cup [|U'\ W^n_{\Sigma k-1}| + 1 \ldots |X| + 1],$$

the following holds:

$$(\alpha_1, j_1) \models_{M,X} (\alpha_2, j_2) \quad \Longleftrightarrow \quad (\alpha_1, \tilde{j}_1) \models_{M,Y} (\alpha_2, \tilde{j}_2).$$

Proof. For suitable $Z_1, Z_2 \in \{0,1\}^*$, the words considered can be written as

$$\begin{aligned} W^n_{\Sigma k} &= Z_1\ 1^n\ Z_2 \text{ and } W^n_{\Pi k} = Z_1\ 1^{n+n!}\ Z_2 && \text{if } k \text{ is odd, and} \\ W^n_{\Sigma k} &= Z_1\ 1^{n+n!}\ Z_2 \text{ and } W^n_{\Pi k} = Z_1\ 1^n\ Z_2 && \text{if } k \text{ is even.} \end{aligned}$$

The claim then follows from the pumping lemma (Lemma 3). □

(A) First we consider existential configurations (α, i). Assume that

$$\mathtt{acc}_M^k(\alpha, i, X)$$

is true. Hence there exists an existential computation path from (α, i) to a final or universal configuration (β, j),

$$\text{(ii)} \qquad (\alpha, i) \models_{M,X} (\beta, j),$$

with the property

$$\text{(iii)} \qquad \mathtt{acc}_M^{k-1}(\beta, j, X).$$

We may assume that

$$\text{(iv)} \qquad j \leq |U'| \quad \text{or} \quad j > |U' \ W_{\Sigma k-1}^n|$$

because if $|U'| < j \leq |U' \ W_{\Sigma k-1}^n|$, then for $Z_1 := U$, $Z_2 := V$, $W := W_{\Sigma k-1}^n \ 0$, and $s := 2m_{k,n} + 1 - n - (n + n!)$, the configuration-shift lemma implies

$$(\alpha, i) \models_{M,X} (\beta, j - \Delta).$$

Moreover, for $r := t := m_{k,n}$ and for $s := 1$, from the position-shift lemma, we can deduce

$$\mathtt{acc}_M^{k-1}(\beta, j - \Delta, X).$$

Therefore, if $|U'| < j \leq |U' \ W_{\Sigma k-1}^n|$, the configuration (β, j') with $j' := j - \Delta$ instead of (β, j) satisfies properties (ii)–(iv).

Since $\hat{i} = \tilde{i}$ according to (i), Claim 4 applied to (ii) yields

$$(\alpha, \hat{i}) = (\alpha, \tilde{i}) \models_{M,Y} (\beta, \tilde{j}).$$

A terminating configuration (β, j) must be accepting because of (ii) and (iii); hence $(\beta, \tilde{j})$ is accepting and $\mathtt{acc}_M^k(\alpha, \hat{i}, Y)$ is true.

For a universal (β, j), we apply the induction hypothesis. Because of (iv), requirements 1 and 2 of the proposition are fulfilled for $k - 1$ and $i := \tilde{j}$. Property (i) implies for this choice of i that $\hat{i} = j$. Therefore, in (iii), replacing j by $\hat{i}$, we conclude

$$\mathtt{acc}_M^{k-1}(\beta, \hat{i}, X) \qquad \Longrightarrow \qquad \mathtt{acc}_M^{k-1}(\beta, i, Y) = \mathtt{acc}_M^{k-1}(\beta, \tilde{j}, Y).$$

Hence we can conclude that $\mathtt{acc}_M^k(\alpha, \hat{i}, Y)$ holds. This proves the proposition for existential configurations.

(B) Now let us consider universal configurations (α, i), for which $\mathtt{acc}_M^k(\alpha, \hat{i}, Y)$ holds. We have to show that $\mathtt{acc}_M^k(\alpha, i, X)$ is true.

CLAIM 5. *For input X, any universal computation path starting in (α, i) is finite.*

Proof. Assume, to the contrary, that there exists an infinite computation path which is universal and starts in (α, i). This means that there exists a universal configuration (β, j) such that

$$\text{(v)} \qquad (\alpha, i) \models_{M,X} (\beta, j) \models_{M,X} (\beta, j).$$

We can assume that

$$\text{(vi)} \qquad j \leq |U'| \quad \text{or} \quad j > |U' \ W_{\Sigma k-1}^n|$$

because if $|U'| < j \le |U'\ W^n_{\Sigma k-1}|$, the configuration-shift lemma implies

$$(\alpha, i) \models_{M,X} (\beta, j - \Delta) \models_{M,X} (\beta, j - \Delta).$$

Hence (v) and (vi) are fulfilled for $j' := j - \Delta$. From (i) and Claim 4, it follows that

$$(\alpha, \hat{i}) = (\alpha, \tilde{i}) \models_{M,Y} (\beta, \tilde{j}) \models_{M,Y} (\beta, \tilde{j}).$$

This means that for input Y there exists an infinite computation path which is universal and starts in $(\alpha, \hat{i})$. We get a contradiction to $\mathsf{acc}^k_M(\alpha, \hat{i}, Y)$. □

Now we want to show that for any final or existential configuration (β, j) that can be reached from (α, i) on a universal computation path, the following holds:

$$\mathsf{acc}^{k-1}_M(\beta, j, X).$$

According to Claim 5 this proves $\mathsf{acc}^k_M(\alpha, i, X)$. Let $(\alpha, i) \models_{M,X} (\beta, j)$. Two cases will be distinguished.

Case 1. $j \le |U'|$ or $j > |U'\ W^n_{\Sigma k-1}|$.

From Claim 4, it follows that

$$(\alpha, \hat{i}) = (\alpha, \tilde{i}) \models_{M,Y} (\beta, \tilde{j}).$$

The assumption $\mathsf{acc}^k_M(\alpha, \hat{i}, Y)$ implies

(vii) $$\mathsf{acc}^{k-1}_M(\beta, \tilde{j}, Y).$$

For a final configuration (β, j), one can conclude from property (vii) that β must be accepting; hence $\mathsf{acc}^{k-1}_M(\beta, j, X)$ holds.

For an existential (β, j), the same implication holds using the induction hypothesis.

Case 2. $|U'| < j \le |U'\ W^n_{\Sigma k-1}|$.

The configuration-shift lemma implies

$$(\alpha, i) \models_{M,X} (\beta, j - \Delta).$$

In the proof of Case 1, it was shown for the configuration $(\beta, j - \Delta)$ that

$$\mathsf{acc}^{k-1}_M(\beta, j - \Delta, X)$$

holds. Using the position-shift lemma, we obtain $\mathsf{acc}^{k-1}_M(\beta, j, X)$. This completes the proof of Proposition 1. □

Next, we will show that the second requirement of the proposition above is always fulfilled.

PROPOSITION 2. *Let $k \ge 2$ and M be an ATM of space complexity S with $S \in o(\log)$. Then there exists a bound $S' \in o(\log)$ such that for all $n \ge \mathcal{N}_{M,S'}$,*

$$Space_M(W^n_{\Pi k}) \le S'(n) \quad and \quad Space_M(W^n_{\Sigma k}) \le S'(n).$$

Proof. The idea of the proof is as follows. If in $W^n_{\Pi k}$ and $W^n_{\Sigma k}$ all substrings generated in the recursive construction which are multiplies of $n!$ are cancelled, then the remaining word has a length $p_k(n)$, which is polynomial in n. Using the small-space-bound lemma (Lemma 5), which shows that a sublogarithmic space-bounded machine M does not notice a difference when an arbitrary block of the input is added $n!$ times, it follows that M must obey a space bound $S(p_k(n))$ on $W^n_{\Pi k}$ and $W^n_{\Sigma k}$. If S grows sublogarithmically in n, so does $S(p_k(n))$.

The technical details of this proof are outlined below. Let

$$V^1_2(n) := 1^n.$$

For $d \geq 3$, define

$$V_d^1(n) := \left[V_{d-1}^1(n)\ 0\right]^{2dn+1},$$

and for $i = 2, \ldots, d-1$,

$$V_d^i(n) := \left[V_{d-1}^{i-1}(n)\ 0\right]^{2m_{d,n}+1}.$$

Also define a sequence of polynomials $p_d(n)$ as follows:

$$p_2(n) := n \qquad \text{and, for } d \geq 3, \qquad p_d(n) := (2dn+1) \cdot (p_{d-1}(n) + 1).$$

Obviously, for any $d \geq 2$ and all n,

$$p_d(n) = |V_d^1(n)|.$$

Let M be an ATM of space complexity S with $S \in o(\log)$. Define $S'(n) := S(p_k(n))$. Obviously, $S' \in o(\log)$. Let n be an integer with $n \geq \mathcal{N}_{M,S'}$.

Since M is S space bounded,

(i) $$Space_M(V_k^1(n)) \leq S(p_k(n)) = S'(n).$$

It is easy to check that for any n and any $i \in [1 \ldots k-2]$, there are words $Z_1, Z_2, \ldots, Z_r$ over the alphabet $\{0\}$, where

$$r := \prod_{t=k-i+2}^{k} 2m_{t,n} + 1$$

(for $i = 1$, take $r := 1$), such that for $W := V_{k-i}^1(n)\ 0$, $a := 2n(k-i) + n + 1$, and $b := 2(k-i+1)$,

$$V_k^i(n) = W^{a+n}\ Z_1\ W^{a+n}\ Z_2 \ldots Z_{r-1}\ W^{a+n}\ Z_r,$$

$$V_k^{i+1}(n) = W^{a+n+bn!}\ Z_1\ W^{a+n+bn!}\ Z_2 \ldots Z_{r-1}\ W^{a+n+bn!}\ Z_r.$$

By the small-space-bound lemma, the following implications hold for $i = 1, \ldots, k-2$:

$$Space_M(V_k^i(n)) \leq S'(n) \qquad \Longrightarrow \qquad Space_M(V_k^{i+1}(n)) \leq S'(n).$$

Therefore, by (i), we obtain that

(ii) $$Space_M(V_k^{k-1}(n)) \leq S'(n).$$

Now let $\hat{W}_{\Sigma k}^n$ denote a word $W_{\Sigma k}^n$ where all substrings $1^{n+n!}$ are reduced to 1^n. Similarly, $\hat{W}_{\Pi k}^n$ is obtained from $W_{\Pi k}^n$. Obviously, by the small-space-bound lemma, $Space_M(\hat{W}_{\Sigma k}^n) \leq S'(n)$ implies $Space_M(W_{\Sigma k}^n) \leq S'(n)$ and $Space_M(\hat{W}_{\Pi k}^n) \leq S'(n)$ implies $Space_M(W_{\Pi k}^n) \leq S'(n)$. The proposition holds since

$$\hat{W}_{\Sigma k}^n = \hat{W}_{\Pi k}^n = V_k^{k-1}(n)$$

and by (ii) the space used by M on input $V_k^{k-1}(n)$ is bounded by $S'(n)$. □

Now we are ready to prove Theorem 8. Let us assume that M is a Σ_kTM accepting $L_{\Pi k}$ in sublogarithmic space S. By Proposition 2, there exists a function $S' \in o(\log)$ such that for any $n \geq \mathcal{N}_{M,S'}$,

$$Space_M(W_{\Pi k}^n) \leq S'(n) \quad \text{and} \quad Space_M(W_{\Sigma k}^n) \leq S'(n).$$

Let n with $n \in \mathcal{F}$ be an integer larger than $\mathcal{N}_{M,S'}$ (such an n exists since $\mathcal{F}$ is infinite). By Lemma 14, $W^n_{\Pi k} \in L_{\Pi k}$; hence M has to accept $W^n_{\Pi k}$, which means that $\mathtt{acc}^k_M(\alpha_0, 0, W^n_{\Pi k})$ is true, where $(\alpha_0, 0)$ is the initial configuration of M. From Proposition 1, we conclude that $\mathtt{acc}^k_M(\alpha_0, 0, W^n_{\Sigma k})$ as well, and hence M accepts $W^n_{\Sigma k}$, which by Lemma 14 does not belong to $L_{\Pi k}$, a contradiction.

In the same way, we can show that if M is a Π_kTM that accepts $L_{\Sigma k}$ in space S, then M accepts $W^n_{\Pi k}$.

3.3. Unbounded number of alternations. Let us now consider ATMs with a nonconstant bounding function A for the number of alternations. The separating results for A-alternation-bounded space classes (Theorem 2) follow from the propositions below.

DEFINITION 8. *Let $A : \mathbb{N} \to \mathbb{N}$ be a function with $A(n) \geq 2$ for all n and define*

$$L_\Sigma(A) := \{X \mid X = W0^r \text{ for some } r \in \mathbb{N} \text{ and } W \in L_{\Sigma k} \text{ for some } k \leq A(|X|)\},$$

$$L_\Pi(A) := \{X \mid X = W0^r \text{ for some } r \in \mathbb{N} \text{ and } W \in L_{\Pi k} \text{ for some } k \leq A(|X|)\}.$$

LEMMA 15. *For any $S \in$ SUBLOG and all functions $A \geq 2$ computable in space S, the following hold:*

$$L_\Sigma(A) \in \Sigma_A Space(S),$$
$$L_\Pi(A) \in \Pi_A Space(S).$$

Proof. On input $X = W0^r$, the machine first computes $a := A(|X|)$ and initializes a counter with that value. It remains to check whether $W \in L_{\Sigma k}$ for some $k \leq a$. This can be done similarly as in the case for fixed k, decrementing the counter each time an alternation has been performed. □

For functions $A, B : \mathbb{N} \to \mathbb{N}$, let $A \leq_* B$ denote that $A(m) \leq B(m)$ for all $m \in \mathbb{N}$ with equality for infinitely many m.

PROPOSITION 3. *For any $S \in$ SUBLOG and all functions A and B with $1 < A \leq_* B$ and $B \cdot S \in o(\log)$, the following hold:*

$$L_\Sigma(A) \notin \Pi_B Space(S),$$
$$L_\Pi(A) \notin \Sigma_B Space(S).$$

Proof. Let $S \in$ SUBLOG and let A and B be functions with $1 < A \leq_* B$ and $B \cdot S \in o(\log)$. These assumptions imply that there exists a constant $m_0 \geq \exp\exp 9$ such that $A(m) < \frac{\log m}{\operatorname{llog} m}$ for all $m \geq m_0$. Define the functions h and f as follows:

$$h(m) := \frac{\exp\left(\frac{\log m}{A(m)}\right)}{3\, A(m)},$$

$$f(m) := \max\{\ell \mid \ell \in \mathcal{F} \cup \{0\},\ \ell \leq h(m)\}.$$

For $m \geq m_0$, we can bound h by

$$h(m) \leq \exp\left(\frac{\log m}{2}\right) = m^{1/2},$$

$$h(m) \geq \frac{\exp \operatorname{llog} m}{3 \log m \ / \ \operatorname{llog} m} = \frac{\operatorname{llog} m}{3} \geq 3,$$

and hence $f(m) \in \mathcal{F}$. Moreover, from Lemma 11, it follows that

(i) $$f(m) \geq h(m)^{1/3} \geq \left(\frac{1}{3} \operatorname{llog} m\right)^{1/3}.$$

Define the function $S' : \mathbb{N} \to \mathbb{N}$ as follows:

$$S'(n) := \max(\{0\} \cup \{S(m) \mid f(m) = n\}).$$

Because f grows unboundedly, $S'(n)$ will always be a finite number.

LEMMA 16. $S' \in o(\log)$.

Proof. First we show that $S \in o(\log \circ f)$. By assumption,

$$S \in o\left(\frac{\log}{A}\right) \qquad \text{and} \qquad \log A \;\leq\; \operatorname{llog} m \;\leq\; S.$$

This implies

$$S \in o\left(\frac{\log}{A} - S\right) \;=\; o\left(\frac{\log}{A} - \log A\right) \;=\; o(\log h) \;=\; o(\log f).$$

Thus if n goes to ∞,

$$\frac{S(n)}{\log f(n)} \to 0$$

and

$$\frac{S'(n)}{\log n} = \max_{\{m \mid f(m)=n\}} \frac{S(m)}{\log n} = \max_{\{m \mid f(m)=n\}} \frac{S(m)}{\log f(m)}.$$

If n goes to ∞, m has to as well, and hence all quotients converge to 0. But this means that $S' \in o(\log)$. □

Consider the function t defined by

$$t(m) := m - p_{A(m)}(f(m)),$$

where $p_d(n)$ has already been defined in the proof of Proposition 2, and note that

$$p_{A(m)}(f(m)) \;\leq\; (3\,A(m)\;f(m))^{A(m)} \;\leq\; m.$$

Thus $t(m) \geq 0$.

Now let M be an ATM that works in space $S(|X|)$ and makes at most $B(|X|) - 1$ alternations. Let m be an integer with

$$\text{(ii)} \qquad m \;\geq\; \max\{m_0,\; \exp\exp 3(\mathcal{N}_{M,S'})^4\} \quad \text{and} \quad A(m) = B(m).$$

Such an m exists since $A \leq_* B$. Then define

$$k := A(m) \quad \text{and} \quad n := f(m).$$

By (i) and (ii), $n \geq \mathcal{N}_{M,S'}$. Moreover, $n \in \mathcal{F}$ and M makes no more then $k - 1$ alternations on any input of length m. Let

$$X := V_k^1(n)\,0^{t(m)},$$

with the word $V_k^1(n)$ defined as in the proof of Proposition 2. Since the length of $V_k^1(n)$ is $p(k, n)$, the string X is of length m. From the definition of S', it follows that

$$\begin{aligned} Space_M(X) &\leq S(m) \;\leq\; \max\{S(m') \mid f(m') = n\} \;=\; S'(n) \text{ and} \\ Alter_M(X) &\leq B(m) - 1 \;\leq\; \exp S(m) \;\leq\; \exp S'(n). \end{aligned}$$

Hence for the machine M and the function S', the assumptions of the small-space-bound and small-alternation-bound lemmas (Lemmas 5 and 6, respectively) are fulfilled. Using the small-space-bound lemma for the input X in the similar way as in the proof of Proposition 2,we can show that

$$Space_M(W^n_{\Sigma k}\, 0^{t(m)}),\ Space_M(W^n_{\Pi k}\, 0^{t(m)}) \;=\; Space_M(X) \leq S'(n).$$

Similarly, by the small-alternation-bound lemma, we obtain

$$Alter_M(W^n_{\Sigma k}\, 0^{t(m)}),\ Alter_M(W^n_{\Pi k}\, 0^{t(m)}) \;=\; Alter_M(X) \leq B(m) - 1 = k - 1.$$

Now we can finish the proof. Let us assume that M is a Σ_BTM accepting $L_\Pi(A)$ in space S. By Lemma 14, $W^n_{\Pi k} \in L_{\Pi k}$ holds; hence M has to accept $W^n_{\Pi k}\, 0^{t(m)}$. But this means that $\mathsf{acc}^k_M(\alpha_0, 0, W^n_{\Pi k}\, 0^{t(m)})$ is true, where $(\alpha_0, 0)$ is the initial configuration of M. From Proposition 1, we conclude that $\mathsf{acc}^k_M(\alpha_0, 0, W^n_{\Sigma k}\, 0^{t(m)})$ holds as well. Therefore M accepts $W^n_{\Sigma k}\, 0^{t(m)}$, which by Lemma 14 does not belong to $L_\Pi(A)$, a contradiction.

In the same way, we can show that if M is a Π_BTM that accepts $L_\Sigma(A)$ in space S, then M accepts $W^n_{\Pi k}\, 0^{t(m)}$. □

4. Closure properties. In this section, we discuss closure properties of the classes $\Sigma_k Space(S)$ and $\Pi_k Space(S)$ for sublogarithmic bounds S. First, for any integer $k \geq 2$, we define the languages

$$A_{\Sigma k} := L_k\,\{0\}\,L_{\Sigma k}, \quad B_{\Sigma k} := L_{\Sigma k}\,\{0\}\,L_k,$$

and, symmetrically,

$$A_{\Pi k} := L_k\,\{0\}\,L_{\Pi k}, \quad B_{\Pi k} := L_{\Pi k}\,\{0\}\,L_k.$$

It is easy to see that

(i) $$A_{\Sigma k}, B_{\Sigma k} \in \Sigma_k Space(\mathrm{llog}) \quad \text{and} \quad A_{\Pi k}, B_{\Pi k} \in \Pi_k Space(\mathrm{llog}).$$

PROPOSITION 4. *For all $k \geq 2$, the following hold:*

$$A_{\Sigma k} \cap B_{\Sigma k} \in \Pi_{k+1} Space(\mathrm{llog}) \setminus \Sigma_{k+1} Space(o(\log)),$$
$$A_{\Pi k} \cup B_{\Pi k} \in \Sigma_{k+1} Space(\mathrm{llog}) \setminus \Pi_{k+1} Space(o(\log)).$$

Proof. It is well known that for any function S, the classes $\Sigma_k Space(S)$ are closed under union, and, symmetrically, the $\Pi_k Space(S)$ are closed under intersection (see, e.g., [25]). Hence by (i), $A_{\Sigma k} \cap B_{\Sigma k} \in \Pi_{k+1} Space(\mathrm{llog})$ and $A_{\Pi k} \cup B_{\Pi k} \in \Sigma_{k+1} Space(\mathrm{llog})$. To prove that $A_{\Sigma k} \cap B_{\Sigma k} \notin \Sigma_{k+1} Space(o(\log))$ and $A_{\Pi k} \cup B_{\Pi k} \notin \Pi_{k+1} Space(o(\log))$, we first modify Proposition 2 in the following way.

PROPOSITION 5. *Let $k \geq 2$ and M be an ATM of space complexity S with $S \in o(\log)$. Then there exists a bound $S'' \in o(\log)$ such that for all $n \geq \mathcal{N}_{M,S''}$ and words $W_1, W_2 \in \{W^n_{\Sigma k}, W^n_{\Pi k}\}$,*

$$Space_M(W_1\, 0\, W_2) \leq S''(n).$$

Proof. Let $S''(n) := S(2p_k(n) + 1)$, where p_k is the polynomial specified in the proof of Proposition 2. It is easy to check that the proof of Proposition 2 generalizes to this situation. □

Let us assume, to the contrary, that $A_{\Sigma k} \cap B_{\Sigma k} \in \Sigma_{k+1} Space(S)$ for some $S \in o(\log n)$. Let M be an S space-bounded Σ_{k+1}TM for $A_{\Sigma k} \cap B_{\Sigma k}$. Choose $n \in \mathcal{F}$ sufficiently large. By Lemma 14, $W^n_{\Sigma k} \in L_{\Sigma k}$; hence M has to accept

$$X = W^n_{\Sigma k}\, 0\, W^n_{\Sigma k},$$

which means that there exists an existential computation path starting in initial configuration $(\alpha_0, 0)$ and ending in a universal configuration (β, j), with

(ii) $$(\alpha_0, 0) \models_{M,X} (\beta, j)$$

and

(iii) $$\mathsf{acc}_M^k(\beta, j, X).$$

(The trivial case where M accepts X without alternation could be handled similarly.) Now let $Y_1 := W_{\Sigma k}^n 0\, W_{\Pi k}^n$ and $Y_2 := W_{\Pi k}^n 0\, W_{\Sigma k}^n$. By Proposition 5 there exists $S'' \in o(\log n)$ such that

$$Space_M(X),\ Space_M(Y_1),\ Space_M(Y_2) \leq S''(n).$$

Therefore, applying Claim 1 and Proposition 1 to (ii) and (iii), respectively, we obtain

$$(\alpha_0, 0) \models_{M,Y_1} (\beta, j) \quad \text{and} \quad \mathsf{acc}_M^k(\beta, j, Y_1)$$

if $j \leq |W_{\Sigma k}^n 0|$ and

$$(\alpha_0, 0) \models_{M,Y_2} (\beta, \hat{j}) \quad \text{and} \quad \mathsf{acc}_M^k(\beta, \hat{j}, Y_2)$$

otherwise, where $\hat{j} = j + |Y_2| - |X|$. Hence M also accepts input Y_1 or Y_2. This yields a contradiction since, by Lemma 14, $Y_1, Y_2 \notin A_{\Sigma k} \cap B_{\Sigma k}$.

Similarly, we can show that if a Π_{k+1}TM accepts $A_{\Pi k} \cup B_{\Pi k}$ within space $S \in o(\log n)$, then it has to reject X, but it also rejects input Y_1 or Y_2, which both belong to $A_{\Pi k} \cup B_{\Pi k}$, a contradiction. □

This result can be applied to prove Theorem 3. For all $k \geq 2$ and any $S \in$ SUBLOG, the following hold:

1. $\Sigma_k Space(S)$ and $\Pi_k Space(S)$ are not closed under complementation.
2. $\Sigma_k Space(S)$ is not closed under intersection.
3. $\Pi_k Space(S)$ is not closed under union.
4. $\Sigma_k Space(S)$ and $\Pi_k Space(S)$ are not closed under concatenation.

Property 1 follows immediately from Lemma 13, Theorem 8, and the following equations: $L_{\Sigma k} = L_k \cap \overline{L}_{\Pi k}$ and $L_{\Pi k} = L_k \cap \overline{L}_{\Sigma k}$, where L_k is the regular language introduced in Definition 6.

By (i), $A_{\Sigma k}, B_{\Sigma k} \in \Sigma_k Space(\text{llog})$ and $A_{\Pi k}, B_{\Pi k} \in \Pi_k Space(\text{llog})$. On the other hand, from Proposition 4, $A_{\Sigma k} \cap B_{\Sigma k} \notin \Sigma_{k+1} Space(o(\log))$ and $A_{\Pi k} \cup B_{\Pi k} \notin \Pi_{k+1} Space(o(\log))$. This proves properties 2 and 3.

Property 4 for Σ_k classes follows from the fact that for any $k \geq 2$, $L_{\Sigma k}\{0\}L_{\Sigma k} = A_{\Sigma k} \cap B_{\Sigma k}$ does not belong to $\Sigma_k Space(o(\log))$, but $L_{\Sigma k} \in \Sigma_k Space(\text{llog})$. To see that $\Pi_k Space(S)$ is not closed under concatenation, define the languages

$$L_k^1 := L_k \cup \{\varepsilon\},$$

where ε denotes the empty string, and

$$L_k^2 := \{w_1 0 w_2 0 \ldots 0 w_p 0 \mid p \in \mathbb{N},\ w_i \in L_{k-1} \text{ and } w_1 \in L_{\Pi k-1}\}.$$

Obviously, both languages belong to $\Pi_k Space(\text{llog})$, but from Theorem 8, it follows that

$$L_k^1 L_k^2 = L_{\Sigma k} \notin \Pi_k Space(o(\log)).$$

5. Lower space bounds for context-free languages.

PROPOSITION 6. $L_{\neq} = \{1^n01^m : n \neq m\} \notin ASpace(o(\log))$.

Proof. Let us assume, to the contrary, that $L_{\neq}$ is recognized by an ATM A in space S for some $S \in o(\log)$. Let $S'(n) := S(2n+1)$. Obviously, $S' \in o(\log)$. Let $\hat{n} := \mathcal{N}_{M,S'}$. Then by the small-space-bound lemma (Lemma 5), for all $k, \ell \geq 0$,

(i) $$Space_A(1^{\hat{n}}01^{\hat{n}}) = Space_A(1^{\hat{n}+k\hat{n}!}01^{\hat{n}+\ell\hat{n}!}).$$

Let

(ii) $$\hat{s} = Space_A(1^{\hat{n}}01^{\hat{n}}).$$

For this fixed $\hat{n}$, we define the language $\hat{L} = \{1^{\hat{n}+k\hat{n}!}01^{\hat{n}+\ell\hat{n}!} : k, \ell \in \mathbb{N} \text{ and } k \neq \ell\}$ and construct an automaton $\hat{A}$ that recognizes $\hat{L}$. $\hat{A}$ performs the following algorithm.

Step 1. Check deterministically if the input X has the form $1^{\hat{n}+k\hat{n}!}01^{\hat{n}+\ell\hat{n}!}$ for some integers k and ℓ; reject and stop if this condition does not hold.

Step 2. Move the head to the first symbol of the input and start to simulate the machine A.

It is obvious that $\hat{A}$ accepts an input $X = 1^{\hat{n}+k\hat{n}!}01^{\hat{n}+\ell\hat{n}!}$ iff A accepts X. Hence we have $L(\hat{A}) = \hat{L}$. It is easy to see that Step 1 can be performed within space $O(\log \hat{n}!)$, which is a constant. Moreover, from (i) and (ii), it follows that Step 2 also requires only constant space $\hat{s}$. Hence $\hat{A}$ recognizes $\hat{L}$ within constant space. We get a contradiction since $\hat{L}$ is nonregular. □

Using a similar proof, we can show that the language

$$L_{=} := \{1^n01^n : n \in \mathbb{N}\}$$

is also not in $ASpace(o(\log))$.

The rest of this section is devoted to the lower space bounds for a large subset of nonregular context-free languages.

The block structure of a bounded language L can equivalently be represented using a finite alphabet $\{a_1, \ldots, a_r\}$. Then L is a subset of $\{a_1\}^* \ldots \{a_r\}^*$.

DEFINITION 9. *Let $V(L)$ denote the set $\{(v_1, \ldots, v_r) \in \mathbb{N}^r \mid a_1^{v_1} \ldots a_r^{v_r} \in L\}$. Sets of the form $\{\alpha + n_1\beta_1 + \cdots + n_k\beta_k \mid n_1, \ldots, n_k \in \mathbb{N}\}$ with $\alpha, \beta_1, \ldots, \beta_k \in \mathbb{N}^r$, are called* linear sets. *A finite union of linear sets is a* semilinear set. *A language L is* semilinear *if $L \subseteq \{a_1\}^* \ldots \{a_r\}^*$ and $V(L)$ is a semilinear set.*

PROPOSITION 7. *Let a language $L \subseteq \{a_1\}^* \ldots \{a_r\}^*$ be semilinear and let $L, \overline{L} \in ASpace(S)$ for some $S \in o(\log)$. Then L is regular.*

Proof. For $r = 1$, the proposition is true because every semilinear tally language is regular. Let us assume that $r > 1$ and that the proposition holds for $r - 1$. Sets of the form

$$\{\alpha + q_1\gamma_1 + \cdots + q_k\gamma_k \mid q_1, \ldots, q_k \in \mathbb{R}_+\}$$

with $\gamma_1, \ldots, \gamma_k \in \mathbb{N}^r$ are called *cones* (see [1]). Now assume, to the contrary, that L is nonregular. To show that this cannot occur, we first construct a semilinear language $\tilde{L} \in ASpace(S)$ that is also nonregular and for which there exists an r-dimensional cone C such that $V(\tilde{L}) \cap C = \emptyset$. To this end, methods developed by Alt and Mehlhorn [1], [4] will be used.

LEMMA (see [1]). *There exists an r-dimensional cone C and a regular language $R \subseteq \{a_1\}^* \ldots \{a_r\}^*$ with*

$$V(L) \cap C = V(R) \cap C.$$

Let R and C be as in the lemma. Define $L_1 := L \setminus R$ and $L_2 := R \setminus L$. Obviously, L_1 or L_2 is nonregular since L is nonregular. We set $\tilde{L} := L_1$ if L_1 is nonregular and $\tilde{L} := L_2$ otherwise. The language $\tilde{L}$ is semilinear since the class of semilinear sets is closed under Boolean operations [12]. Moreover, $\tilde{L} \in ASpace(S)$, because $L, \overline{L} \in ASpace(S)$, and $V(\tilde{L}) \cap C = \emptyset$ for the r-dimensional cone C.

DEFINITION 10. *Let us call a set* $K \subseteq \mathbb{N}^r$ extended *if there exists* $\alpha \in \mathbb{N}^r$ *and* $\beta \in \mathbb{N}_+^r$ *such that*

$$\forall\, k \in \mathbb{N} \quad \alpha + k\beta \in K.$$

Remark. In [1], a different definition of extended set has been used. However, it is easy to check that both definitions are equivalent.

If $V(\tilde{L})$ is not extended, then one can show similarly as in [1] that there exists a nonregular language in $\{a_1\}^* \dots \{a_{r-1}\}^*$ fulfilling the assumptions of the proposition. Hence by the inductive hypothesis, we obtain a contradiction. Therefore, we can assume that $V(\tilde{L})$ is extended. Let $\alpha = (\alpha_1, \dots, \alpha_r)$ and $\beta = (\beta_1, \dots, \beta_r)$, with $\alpha_1, \dots, \alpha_r \in \mathbb{N}$ and $\beta_1, \dots, \beta_r \in \mathbb{N}_+$, be vectors such that

$$\forall\, k \in \mathbb{N} \quad \alpha + k\beta \in V(\tilde{L}).$$

Moreover, let $\tilde{M}$ be an ATM which recognizes $\tilde{L}$ in space S. Define the function S' as

$$S'(n) := S\left(\sum_{i=1}^{r} \alpha_i \;+\; n \sum_{i=1}^{r} \beta_i\right).$$

Since $S \in o(\log)$, $S' \in o(\log)$ also. Let $\hat{n} := \mathcal{N}_{\tilde{M},S'}$. Then we define

$$\hat{R} := \{a_1^{\alpha_1+(\hat{n}+\ell_1\hat{n}!)\beta_1} \dots a_r^{\alpha_r+(\hat{n}+\ell_r\hat{n}!)\beta_r} \mid \ell_1, \dots, \ell_r \in \mathbb{N}\} \quad \text{and} \quad \hat{L} := \hat{R} \cap \tilde{L}.$$

A contradiction will be obtained from the following claims

CLAIM 6. *$\hat{L}$ can be recognized in constant space.*

CLAIM 7. *$\hat{L}$ is nonregular.*

Proof of Claim 6. Using for every $i = 1, \dots, r$ β_i-times the small-space-bound lemma we obtain that for any sequence of integers $\ell_i \geq 0$

$$\text{(i)} \qquad \hat{s} := Space_{\tilde{M}}(a_1^{\alpha_1+(\hat{n}+\ell_1\hat{n}!)\beta_1} \dots a_r^{\alpha_r+(\hat{n}+\ell_r\hat{n}!)\beta_r}) = S'(a_1^{\alpha_1+\hat{n}\beta_1} \dots a_r^{\alpha_r+\hat{n}\beta_r}).$$

Let $\hat{M}$ be an ATM which performs the following algorithm.

Step 1. Check deterministically if the input X has the form

$$a_1^{\alpha_1+(\hat{n}+\ell_1\hat{n}!)\beta_1} \dots a_r^{\alpha_r+(\hat{n}+\ell_r\hat{n}!)\beta_r}$$

for some integers $\ell_1, \dots, \ell_r$. Reject and stop if this condition does not hold.

Step 2. Move the head to the first symbol of the input and start to simulate the machine $\tilde{M}$.

It is obvious that $\hat{M}$ accepts an input $X = a_1^{\alpha_1+(\hat{n}+\ell_1\hat{n}!)\beta_1} \dots a_r^{\alpha_r+(\hat{n}+\ell_r\hat{n}!)\beta_r}$ iff $\tilde{M}$ accepts X. Hence we have $L(\hat{M}) = \hat{L}$. It is easy to see that Step 1 can be performed within space $O(\log \hat{n}!)$, which is a constant. Moreover, from (i), it follows that Step 2 also requires only constant space $\hat{s}$. Hence $\hat{M}$ recognizes the language $\hat{L}$ within constant space. □

Proof of Claim 7. A set of the form

$$\{\gamma + (k_1\delta_1, \dots, k_r\delta_r) \mid k_1, \dots, k_r \in \mathbb{N}\}$$

with $\gamma \in \mathbb{N}^r$ and $\delta_1, \ldots, \delta_r \in \mathbb{N}$ is called a *grid*. We show that if $\hat{L}$ is regular then there exists an r-dimensional grid in $V(\tilde{L})$.

Assume that $\hat{L}$ is regular. Then, using the pumping lemma (Lemma 3) for regular languages, one can show that there exist integers $\ell \geq 0$ and $\delta_1, \ldots, \delta_r > 0$ such that for all $k_1, \ldots, k_r \geq 0$,

$$\alpha + (\hat{n} + \ell\hat{n}!)\beta + (k_1\delta_1, \ldots, k_r\delta_r) \in V(\hat{L}).$$

Therefore, the r-dimensional grid $G := \{\gamma + (k_1\delta_1, \ldots, k_r\delta_r) \mid k_1, \ldots, k_r \in \mathbb{N}\}$, where $\gamma = \alpha + (\hat{n} + \ell\hat{n}!)\beta$, is a subset of $V(\hat{L})$, which implies $G \subseteq V(\tilde{L})$. From this and the property $V(\tilde{L}) \cap C = \emptyset$ shown above, we obtain that $G \cap C = \emptyset$ for the r-dimensional cone C. This yields a contradiction to the following result.

LEMMA (see [1]). *Let $G \subseteq \mathbb{N}^r$ be an r-dimensional grid and let $C \subseteq \mathbb{N}^r$ be an r-dimensional cone. Then $G \cap C \neq \emptyset$.*

This completes the proof of Proposition 7. □

Recall that a language L is called *strictly nonregular* if there are strings u, v, w, x, and y such that $L \cap \{u\}\{v\}^*\{w\}\{x\}^*\{y\}$ is context free and nonregular. It was shown by Stearns [20] that every nonregular deterministic context-free language is strictly nonregular. Therefore, from Proposition 7, we immediately obtain that if L is a nonregular deterministic context-free, strictly nonregular, or nonregular context-free bounded language, then for ATMs without any bound on the number of alternations, it is not possible that L and $\overline{L}$ both belong to $ASpace(o(\log))$. Moreover, from Theorem 7, it follows that the class of languages recognized by space-bounded ATMs with a constant number of alternations is closed under complement. Hence it follows that the language L does not belong to $\bigcup_{k\in\mathbb{N}} \Sigma_k Space(o(\log))$. This completes the proof of Theorem 6.

6. Conclusions.

The obvious question remaining is how classes $\Sigma_1 Space(S)$ and $\Pi_1 Space(S)$ compare. It is somewhat annoying that the techniques developed in this paper do not provide any help for the case $k = 1$. It is not completely unrealistic to believe that both classes may be equal, which would give the novel result that a hierarchy is infinite although its first level collapses.

If we restrict the complexity classes to bounded languages, $\Sigma_1 Space(S)$ is closed under complementation and both classes are identical, which has been shown in [2] and [23]. But for $k = 2$, the situation changes completely. The languages $L_{\Sigma 2}$ and $L_{\Pi 2}$ are unary—the most stringent form of a bounded language—and still separate $\Sigma_2 Space(S)$ from $\Pi_2 Space(S)$. Thus a separation of the first level would require syntactically more complex languages than the second level. For $k > 2$, the languages $L_{\Sigma k}$ and $L_{\Pi k}$ used in this paper to establish the separation are no longer bounded. But by Proposition 4, the third level can also be separated using the simple bounded languages $A_{\Sigma 2} \cap B_{\Sigma 2}$ and $A_{\Pi 2} \cup B_{\Pi 2}$ that both are subsets of $\{1\}^*\{0\}\{1\}^*$.

Nothing seems to be known for level 4 and higher. Thus the sublogartihmic space hierarchy for bounded languages may be even more complex. We have made some observations leading to the conjecture that for bounded languages this hierarchy might indeed consist of only a finite number of distinct levels.

Finally, it would be nice to characterize the exact relationship between classes $co\text{-}\Sigma_k Space(S)$ and $\Pi_k Space(S)$ for sublogarithmic space bounds S and the class of arbitrary languages.

REFERENCES

[1] H. ALT, *Lower bounds on space complexity for context-free recognition*, Acta Inform., 12 (1979), pp. 33–61.

[2] H. ALT, V. GEFFERT, AND K. MEHLHORN, *A lower bound for the nondeterministic space complexity of context-free recognition*, Inform. Process. Lett., 42 (1992), pp. 25–27.

[3] H. ALT AND K. MEHLHORN, *A language over a one symbol alphabet requiring only $O(\log \log n)$ space*, SIGACT Newslett., 1975, pp. 31–33.

[4] ———, *Lower bounds for the space complexity of context free recognition*, in Proc. 3rd International Colloquium on Automata, Languages, and Programming (ICALP), Lecture Notes in Comput. Sci., Springer-Verlag, Berlin, 1976, pp. 339–354.

[5] B. VON BRAUNMÜHL, *Alternation for two-way machines with sublogarithmic space*, Proc. 10th Symposium on Theoretical Aspects of Computer Science, Lecture Notes in Comput. Sci., Springer-Verlag, Berlin, 1993, pp. 5–15.

[6] B. VON BRAUNMÜHL, R. GENGLER, AND R. RETTINGER, *The alternation hierarchy for machines with sublogarithmic space is infinite*, Research report, Universität Bonn, Bonn, Germany, January, 1993.

[7] J. CHANG, O. IBARRA, B. RAVIKUMAR, AND L. BERMAN, *Some observations concerning alternating Turing machines using small space*, Inform. Process Lett., 25 (1987), pp. 1–9.

[8] V. GEFFERT, *Nondeterministic computations in sublogarithmic space and space constructability*, SIAM J. Comput., 20 (1991), pp. 484–498.

[9] ———, *Sublogarithmic Σ_2-space is not closed under complement and other separation results*, Technical report, University of P. J. Šafárik, Košice, Slovakia, 1992.

[10] ———, *Tally version of the Savitch and Immerman–Szelepcsényi theorems for sublogarithmic space*, SIAM J. Comput., 22 (1993), pp. 102–113.

[11] ———, *A hierarchy that does not collapse: Alternations in low level space*, manuscript.

[12] S. GINSBURG, *The mathematical theory of context-free languages*, McGraw–Hill, New York, 1972.

[13] N. IMMERMAN, *Nondeterministic space is closed under complementation*, SIAM J. Comput., 17 (1988), pp. 935–938.

[14] M. LIŚKIEWICZ AND R. REISCHUK, *Separating the lower levels of the sublogarithmic space hierarchy*, Technical report, Institut für Theoretische Informatik, Technische Hochschule Darmstadt, 1992; Proc. 10th Symposium on Theoretical Aspects of Computer Science, Lecture Notes in Comput. Sci., Springer-Verlag, Berlin, 1993, pp. 16–27.

[15] ———, *The sublogarithmic space hierarchy is infinite*, Technical report, Institut für Theoretische Informatik, Technische Hochschule Darmstadt, Darmstadt, Germany, 1993.

[16] B. LITOW, *On efficient deterministic simulation of Turing machine computations below logspace*, Math. Systems Theory, 18 (1985), pp. 11–18.

[17] P. MICHEL, *A survey of space complexity*, Theoret. Comput. Sci., 101 (1992), pp. 99–132.

[18] D. RANJAN, R. CHANG, AND J. HARTMANIS, *Space bounded computations: Review and new separation results*, Theoret. Comput. Sci., 80 (1991), pp. 289–302.

[19] M. SIPSER, *Halting space-bounded computations*, Theoret. Comput. Sci., 10 (1980), pp. 335–338.

[20] R. E. STEARNS, *A regularity test for pushdown-machines*, Inform. and Control, 11 (1967), pp. 323–340.

[21] R. E. STEARNS, J. HARTMANIS, AND P. M. LEWIS, *Hierarchies of memory limited computations*, in Proc. 1965 IEEE Conference Record on Switching Circuit Theory and Logical Design, IEEE Press, Piscataway, NJ, 1965, pp. 179–190.

[22] R. SZELÉPCSENYI, *The method of forced enumeration for nondeterministic automata*, Acta Inform., 26 (1988), pp. 279–284.

[23] A. SZEPIETOWSKI, *Turing Machines with Sublogarithmic Space*, Lecture Notes on Comput. Sci. 843, Springer-Verlag, Berlin, New York, Heidelberg, 1994.

[24] K. WAGNER, *The alternation hierarchy for sublogarithmic space: An exciting race to STACS* '93 (editiorial note), in Proc. 10th Symposium on Theoretical Aspects of Computer Science, Lecture Notes in Comput. Sci., Springer-Verlag, Berlin, 1993, pp. 2–4.

[25] K. WAGNER AND G. WECHSUNG, *Computational Complexity*, Reidel, Dordrech, 1986.

SIAM J. COMPUT.
Vol. 25, No. 4, pp. 862–873, August 1996

TREE-ADJOINING LANGUAGE PARSING IN $o(n^6)$ TIME*

SANGUTHEVAR RAJASEKARAN†

Abstract. In this paper, we present algorithms for parsing general tree-adjoining languages (TALs). Tree-adjoining grammars (TAGs) have been proposed as an elegant formalism for natural languages. It was an open question for the past ten years as to whether TAL parsing can be done in time $o(n^6)$. We settle this question affirmatively by presenting an $O(n^3\ M(n))$-time algorithm, where $M(k)$ is the time needed for multiplying two Boolean matrices of size $k \times k$ each. Since $O(k^{2.376})$ is the current best-known value for $M(k)$, the time bound of our algorithm is $O(n^{5.376})$. On an exclusive-read exclusive-write parallel random-access machine (EREW PRAM), our algorithm runs in time $O(n \log n)$ using $(n^2 M(n))/\log n$ processors. In comparison, the best-known previous parallel algorithm had a run time of $O(n)$ using n^5 processors (on a systolic-array machine).

We also present algorithms for parsing context-free languages (CFLs) and TALs whose worst-case run times are $O(n^3)$ and $O(n^6)$, respectively, but whose average run times are better. Therefore, these algorithms may be of practical interest.

Key words. tree-adjoining languages, parsing, natural language processing, parallel algorithms, context-free grammars

AMS subject classifications. 11Y16, 68N20, 68Q20, 68Q22, 68Q25, 68S05, 68U30

1. Introduction.

In [9], Joshi, Levy, and Takahashi introduced a grammatical formalism called tree-adjoining grammar (TAG). TAGs are strictly more expressive than context-free grammars (CFGs). For instance, $\{a^n b^n c^n | n \geq 0\}$ can be generated with a TAG, but this language is not context free. TAGs have been shown to be good grammatical systems for natural languages [10]. In this paper, we will restrict our discussion only to the problem of language recognition, since such an algorithm can also be used for retrieving a parse. We use the terms "parsing" and "recognition" interchangeably.

Many papers have been written on the parsing problem for TAGs. Vijayashanker and Joshi [19] gave the first polynomial-time algorithm for tree-adjoining language (TAL) parsing. Their algorithm had a run time of $O(n^6)$, assuming that the size of the grammar is constant. This assumption has been made in most of the literature on parsing. This algorithm had a flavor similar to that of the Cocke–Kasami–Younger (CKY) algorithm for context-free language (CFL) parsing.

An Earley-type parsing algorithm has been given by Schabes and Joshi [16]. This algorithm also takes $O(n^6)$ time. For some special cases of TALs, better algorithms have been discovered [15]. Several attempts have been made in the past to obtain $o(n^6)$-time algorithms (see, e.g., [5, 6]). The parallel complexity of TAL parsing has been studied as well. As an example, Palis, Shende, and Wei present an optimal linear-time algorithm on a systolic-array machine with n^5 processing elements [12]. An excellent treatise on TAGs can be found in [13]. Recently, Nurkkala and Kumar [11] presented an efficient $O(n^6)$-work parallel algorithm for TAL parsing.

It was an open question since 1986 [19] as to whether general TAL parsing can be done in $o(n^6)$ time. In this paper, we present an algorithm that runs in time $O(n^3\ M(n))$, where $M(k)$ is the time needed for multiplying two Boolean matrices of size $k \times k$ each. The best-known value for $M(k)$ is $O(k^{2.376})$ as has been shown by Coppersmith and Winograd [2]. In a related work, Valiant [18] has shown that CFL parsing can be reduced to Boolean

*Received by the editors December 15, 1993; accepted for publication (in revised form) December 22, 1994. This research was supported in part by NSF Research Initiation Award CCR-92-09260 and ARO grant DAAL03-89-C-0031.

†Department of Computer and Information Sciences, University of Florida, P.O. Box 116120, Gainesville, FL 32611.

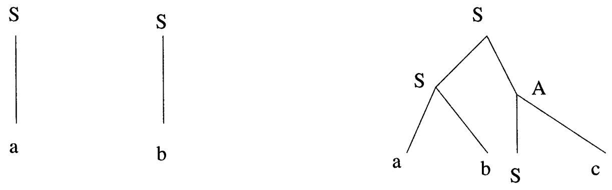


FIG. 1. *TAGs: An example.*

matrix multiplication. Similar work for CFL parsing has also been done by Graham, Harrison, and Ruzzo [4]. Although we make use of algorithms for Boolean matrix multiplication, our approach is different from Valiant's. Direct application of Valiant's technique to TAL parsing seems to fail [14, 15].

Our TAL-parsing algorithm also runs in $O(n \log n)$ time using $(n^2 M(n))/\log n$ exclusive-read exclusive-write parallel random-access machine (EREW PRAM) processors. This run time is nearly the same as that of the previously best-known algorithm for TAL parsing [12]. The processor bound of our algorithm is significantly better than that of [12]. Although we make use of a different model than that of [12], for example, one could invoke Ranade's PRAM-simulation algorithm to infer that it is possible to obtain nearly the same run time on the systolic-array machine, nearly preserving the work done.

2. Definition of TAGs. A TAG is a 5-tuple $G = (N, \Sigma \cup \{\epsilon\}, I, A, S)$, where
N is a finite set of nonterminal symbols,
Σ is a finite set of terminal symbols disjoint from N,
ϵ is the empty terminal string not in Σ,
I is a finite set of labeled *initial trees*,
A is a finite set of labeled *auxiliary trees*, and
$S \in N$ is the distinguished start symbol.

Initial and auxiliary trees of a TAG are *elementary trees*. All internal nodes of elementary trees are labeled by nonterminal symbols. Every initial tree is labeled at the root by the start symbol S and has leaf nodes labeled by symbols from $\Sigma \cup \{\epsilon\}$. An auxiliary tree has both its root and exactly one leaf node (called the *foot node*) labeled by the same nonterminal symbol. All other leaf nodes are labeled by symbols from $\Sigma \cup \{\epsilon\}$. An example of a TAG is given in Figure 1.

An operation called *adjunction* composes trees of the grammar as follows: Let γ be a tree containing some internal node labeled X, and let β be an auxiliary tree whose root is also labeled by X. Adjoining β into γ results in the tree α (see Figure 2). The formalism also supports *constrained adjunction, selective adjunction, and obligatory adjunction.* See, e.g., [19].

Figure 4 in §4 displays a TAG that generates the language $L = \{a^n b^n c^n | n \geq 0\}$. Using the pumping lemma for CFGs, we could readily verify that L is not context free. Thus TAGs are strictly more powerful than CFGs. Also, TAGs possess numerous interesting linguistic properties. The linguistic significance of TAGs is described in [8] and [10].

Joshi [8] showed how TAGs factor recursion and the domain of dependencies in an elegant way. Since the introduction of TAGs, several other formalisms have been proposed for natural languages. Examples include the linear indexed grammars [3] and combinatorial categorial grammars [17]. These formalisms are different in terms of the formal objects and operations defined and were motivated by different aspects of language structure. Surprisingly, all these formalisms have been shown to be equivalent, thereby increasing the possibility that these formalisms capture some of the important and fundamental aspects of languages.

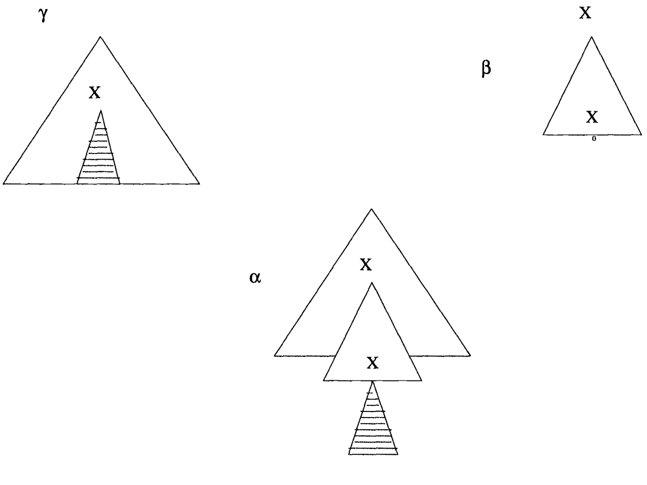


FIG. 2. *The operation of adjunction.*

2.1. Organization of this paper. The rest of this paper is organized as follows. In §3, we provide an overview of Vijayashanker–Joshi's algorithm for TAL parsing. In §4, we present our algorithm for CFL parsing and show how this algorithm can be extended to TALs to obtain a run time of $O(n^6)$. We apply matrix-multiplication algorithms in §5 to show that TAL parsing can be done in time $O(nM(n^2))$. We reduce this run time further to $O(n^3M(n))$ in §6. Finally, §7 concludes the paper.

3. Vijayashanker–Joshi's algorithm. In this section, we provide a brief summary of Vijayashanker–Joshi's algorithm [19]. Their algorithm is similar to the CKY algorithm for CFL parsing. If $a_1a_2 \ldots a_n$ is any given input, the CKY algorithm for CFL parsing constructs a two-dimensional array A such that $A[i, j]$ is the set of all nonterminals that can derive the substring $a_{i+1}a_{i+2} \ldots a_j$. Array A can be constructed in $O(n^3)$ time, and then we simply have to check if the start symbol is in $A[0, n]$.

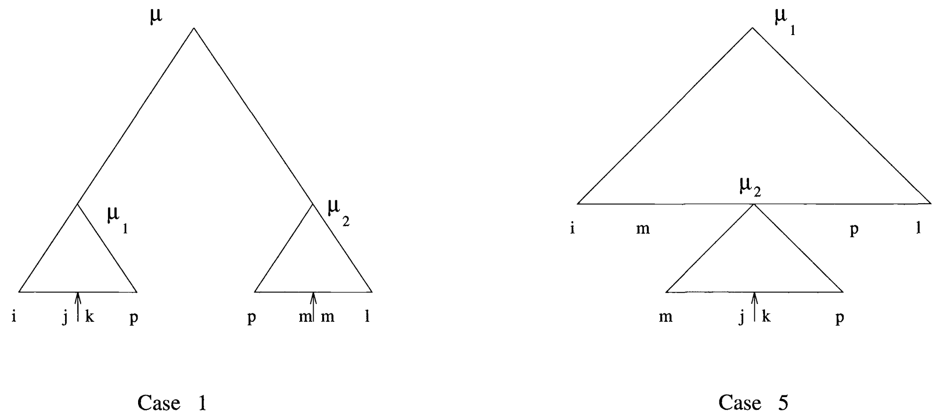
A similar construction is employed in Vijayashanker–Joshi's algorithm [19]. Vijayashanker–Joshi's algorithm [19] is more complicated than the CKY algorithm since a TAG permits the operation of adjunction (which is not in a CFG). A four-dimensional array A is used for TAL parsing.

We assume that the TAG is in normal form, i.e., each node has at most two children. The array A is defined as follows: A node named α is in $A[i, j, k, l]$ if and only if there is a derived tree rooted at α whose *frontier* is given by $a_{i+1}a_{i+2} \ldots a_jYa_{k+1}a_{k+2} \ldots a_l$, where Y is a foot node. The frontier of any tree is defined to be the sequence of labels in the leaves of the tree from left to right.

To begin with, $A[i, i+1, i+1, i+1]$ consists of the nodes in the frontier of elementary trees whose label is a_{i+1}, for $0 \leq i \leq (n-1)$. For all $i \leq j$, $A[i, i, j, j]$ contains the foot nodes of all the auxiliary trees.

The algorithm runs in n^4 *phases*, where each phase consists of the following five cases:

1. If a node μ_1 is in $A[i, j, k, p]$ and another node μ_2 is in $A[p, m, m, l]$ (for some $k \leq p \leq m$ and $p \leq m \leq l$) such that μ_1 and μ_2 are the left and right children, respectively, of node μ, then the node μ belongs to $A[i, j, k, l]$ if μ_1 is the ancestor of the foot node (see Figure 3).
2. This step is symmetric to step 1. Here there are two nodes μ_1 and μ_2 such that $\mu_1 \in A[i, m, m, p]$, $\mu_2 \in A[p, j, k, l]$ (for some $i \leq m \leq p$ and $m \leq p \leq j$), and



Case 1

Case 5

FIG. 3. *Vijayashanker–Joshi's algorithm.*

these two nodes are the left and right children of node μ. In this case, node μ will belong to $A[i, j, k, l]$.

3. If μ is the parent of $\mu_1 \in A[i, j, j, k]$ and $\mu_2 \in A[k, m, m, l]$, then μ belongs to $A[i, j, j, l]$.
4. If μ's only child is $\mu_1 \in A[i, j, k, l]$, then μ is also in $A[i, j, k, l]$.
5. If node μ_2 is in $A[m, j, k, p]$ and the root $\mu_1 \in A[i, m, p, l]$ of some derived tree α has the same label as that of μ_2, then we could adjoin α at μ_2. Thus μ_1 belongs to $A[i, j, k, l]$ (see Figure 3).

Clearly, each phase of the algorithm takes $O(n^2)$ time and there are n^4 entries to fill in the array A. Thus the whole algorithm runs in $O(n^6)$ time. Finally, the given input will be in the language if and only if some initial tree is in $A[0, j, j, n]$, $0 \leq j \leq n$. If the grammar size is also taken into account, this algorithm runs in time $O(n^6\ |G|^2)$ with space complexity $O(n^4\ |G|)$. However, the time bound can be reduced to $O(n^6\ |G| \log |G|)$.

4. The new algorithm. In this section, we present an algorithm for CFL parsing that runs in $O(n^3)$ time with a better average run time. Then we extend this algorithm to TAL parsing.

4.1. CFL parsing. There are two well-known algorithms for CFL parsing, namely, the CKY algorithm and Earley's algorithm, both of which run in time $O(n^3)$. Furthermore, CFL parsing can be reduced to Boolean matrix multiplication [18]. In this subsection, we present an $O(n^3)$-time algorithm that is slightly different from the CKY algorithm. A special feature of this algorithm is that it can be adapted (with some crucial modifications) to parse TALs.

Consider a CFG $G = (N, T, P, S)$ in Chomsky normal form. Each production in P is of the form $A \rightarrow BC$ or $A \rightarrow a$, where A, B, and C are nonterminals and a is a terminal symbol. The basic idea behind the algorithm is the following: The algorithm runs in stages, where in each stage we scan through each production in P and grow larger and larger parse trees. In particular, at any time, each nonterminal has a list of tuples of the form (i, j). If A is any nonterminal, $LIST(A)$ will have tuples (i, j) such that A derives $a_{i+1}a_{i+2}\dots a_j$.

If $A \rightarrow BC$ is a production in P, then in any stage we process this production as follows: We scan through elements in $LIST(B)$ and look for matches in $LIST(C)$. For example, if (i, k) is in $LIST(B)$, we check if (k, j) is in $LIST(C)$, for some j. If so, we insert (i, j) into $LIST(A)$ (if it is not already there). Processing a single production can be done in $O(n^3)$ time if we maintain the following data structures: (1) for each nonterminal A, an array (call it X_A) of lists indexed 1 through n, where $X_A[i]$ is the list of all tuples from $LIST(A)$ whose first item is i ($1 \leq i \leq n$); and (2) an $n \times n$ matrix M whose (i, j)th entry will be those nonterminals that derive $a_{i+1}a_{i+2} \ldots a_j$. There can be $O(n^2)$ entries in $LIST(B)$, and for each entry (i, k) in this list, we need to search for at most n items in $LIST(C)$. Thus the total time needed to process a production is $O(n^3)$.

By induction, we can show that at the end of stage ℓ ($1 \leq \ell \leq n$), the algorithm would have computed all the nonterminals that span any input segment of length ℓ or less. (We say a nonterminal spans the input segment $I = a_{i+1}a_{i+2} \ldots a_j$ if it derives I; the nonterminal is said to have a "span-length" of $j - i$.) Therefore, the algorithm terminates after n stages, implying that the total run time is $O(n^4)$.

However, we can reduce the run time of each stage to $O(n^2)$ as follows: In stage ℓ, while processing the production $A \rightarrow BC$, work only with tuples from $LIST(B)$ and $LIST(C)$ whose combination will derive an input segment of length exactly ℓ. For example, if (i, k) is a tuple in $LIST(B)$, the only tuple in C we should look for is $(k, i + \ell)$. We can look for such a tuple in $O(1)$ time using the matrix M. With this modification, each stage of the above algorithm will only take $O(n^2)$ time, yielding the following

LEMMA 4.1. *This algorithm for CFL parsing runs in time $O(n^3)$ with space complexity $O(n^2)$. The algorithm can also construct parse trees while recognizing CFLs.*

Note. This algorithm has an average run time different from the worst-case run time. For example, if m is the number of elements that will ever be stored in the matrix M, then the run time of the above algorithm is no more than $O(mn)$. In fact, this bound can further be tightened using the fact that a rule of the form $A \rightarrow BC$ can be processed in time $\min\{|LIST(B)|, |LIST(C)|\}$ by keeping track of the length of each list.

On the other hand, for the CKY algorithm, the worst-case and average-case run times are the same. It is well known that Earley's algorithm also has different average and worst-case run times. Perhaps the performance of our variant of the CKY algorithm compares favorably to that of the CKY algorithm. Also, our algorithm has a run time that grows quadratically with the size of the grammar G, just like the CKY algorithm. However, the run time can be reduced to $O(n^3|G| \log |G|)$ by maintaining each entry of the $n \times n$ array as a red–black tree (or as any other balanced binary tree). The $n \times n$ array itself can be replaced with a red–black tree to save space, but the time bound will increase by a logarithmic factor. A comparison between Earley's algorithm and ours will be interesting to investigate. A crucial difference between the two is that our algorithm is bottom-up and Earley's is top-down. For a description of Earley's algorithm, see [1].

In related works, Valiant [18] has shown that CFL parsing can be reduced to Boolean matrix multiplication. Similar work for CFL parsing has also been reported by Graham, Harrison, and Ruzzo [4]. For an excellent treatise on CFL parsing, see [1].

4.2. Extension to TALs. In this subsection, we show how to extend the above algorithm to parsing TALs. The first algorithm we present will have a run time of $O(n^7)$. In §4.5, we will show how to reduce the run time to $O(n^6)$. We keep track of four indices i, j, k, and l corresponding to two different input segments that any derivation tree might span, with a foot node separating these segments, as in Vijayashanker–Joshi's algorithm [19].

We will adapt the algorithm of the previous section. However, the modified algorithm is complicated by the presence of the adjoin operation. The following assumptions (which have been made in all the existing TAL-parsing algorithms) are in place: (1) each node in any tree has ≤ 2 children; (2) each auxiliary tree has at least one terminal symbol in its frontier. (However, assumption (2) can be relaxed.) As in the CFL-parsing algorithm, here also we associate a list with each node in each elementary tree. For any node α, $LIST(\alpha)$ will at any time contain quadruples of the form (i, j, k, l) such that the node spans $a_{i+1} \ldots a_j$ and $a_{k+1} \ldots a_l$ with a nonterminal in between. One of the basic operations that the algorithm performs is

EVALUATE-NODE. This procedure takes as input a tree node—say γ—and computes $LIST(\gamma)$, given the $LIST$s of its children. This procedure composes the two $LIST$s associated with γ's children.

```
EVALUATE-NODE($\gamma$)
/* $\gamma$ is any node in elementary trees. This procedure computes
$LIST(\gamma)$, given the $LIST$s of $\gamma$'s children. */
1   if $\gamma$ has no children then
2       do nothing
3   if $\gamma$ has one child, say $\alpha$, then
4       $LIST(\gamma) := LIST(\alpha)$
5   if $\gamma$ has two children, say $\alpha$ and $\beta$, then
6       for every $(i, j, k, l) \in LIST(\alpha)$ do
7           for every $l \leq m \leq p \leq n$ do
8               if $(l, m, m, p) \in LIST(\beta)$ and $(i, j, k, p) \notin LIST(\gamma)$ then
9                   insert $(i, j, k, p)$ into $LIST(\gamma)$
10          end for
11      end for
12      for every $(l, m, p, q) \in LIST(\beta)$ do
13          for every $1 \leq i \leq j \leq l$ do
14              if $(i, j, j, l) \in LIST(\alpha)$ and $(i, m, p, q) \notin LIST(\gamma)$ then
15                  insert $(i, m, p, q)$ into $LIST(\gamma)$
16          end for
17      end for
```

Here we also maintain an $n^2 \times n^2$ array—say Γ. $\Gamma[i, j, k, l]$ will store all the nodes in elementary trees that are known to span the input segments $a_{i+1} \ldots a_j$ and $a_{k+1} \ldots a_l$, $1 \leq i, j, k, l \leq n$. In this paper, $1 \leq i, j, k, l \leq n$ is shorthand for $1 \leq i \leq j \leq k \leq l \leq n$. With such a data structure, we could perform EVALUATE-NODE(γ) for any node γ in time $O(n^6)$, since there can be no more than n^4 elements in the $LIST$ of any node. Lines 1–4 of EVALUATE-NODE correspond to Step 4 of Vijayashanker–Joshi's algorithm. The rest of the lines in EVALUATE-NODE correspond to Steps 1–3 of their algorithm.

We also employ a procedure called EVALUATE-TREE which works in a bottom-up fashion, evaluating nodes at each level. Basically, this procedure updates the $LIST$ of every node. If γ is an adjoin node labeled X, we also perform ADJOIN(γ, X).

The procedure ADJOIN(α, X) performs adjunction at the node α, where X is the label of the node α. ADJOIN corresponds to Step 5 of Vijayashanker–Joshi's algorithm.

Finally, we have the main procedure called TAL-PARSE, where we evaluate each auxiliary tree and then each initial tree.

```
EVALUATE-TREE($T$)
/* This procedure "EVALUATES" all the nodes of tree $T$ level by level
starting from the bottommost level. */
for each node $\gamma$ in $T$ do
    EVALUATE-NODE($\gamma$)
    if $\gamma$ is an "adjoin" node labeled $X$ then
        ADJOIN($\gamma$, $X$)
end for
```

```
ADJOIN($\alpha$, $X$)
/* This procedure performs the adjoin operation at node $\alpha$ that is labeled by
the nonterminal $X$; $R$ is an array of $n^2$ lists.
$\beta$ is an auxiliary tree labeled by $X$. */
1   Scan through $LIST(\alpha)$ and for each quadruple of the form
    $(q, j, k, r)$ encountered, add $(j, k)$ to the list $R[q, r]$
2   for each $(i, q, r, l) \in LIST(\beta)$ do
        for each $(j, k) \in R[q, r]$ do
            Put $(i, j, k, l)$ into $LIST(\alpha)$ if it is not there
```

```
TAL-PARSE
/* This algorithm takes as input a TAG $G$ and an input $a_1 a_2 \ldots a_n$ and
returns "YES" if the input string is in $L(G)$ and "NO" otherwise. */
0   Initialize the array $\Gamma$ and $LIST(\gamma)$ for each node $\gamma$ in elementary trees
1   for $\ell := 1$ to $n$ do
2       for each auxiliary tree $\gamma$ do
3           EVALUATE-TREE($\gamma$)
4       end for
5       for each initial tree $\gamma$ do
6           EVALUATE-TREE($\gamma$)
7       end for
8   end for
9   if the root of an initial tree $\in \Gamma[0, j, j, n]$ for some $0 \le j \le n$ then
10      return YES else return NO
```

Note. The need to perform EVALUATE-TREE on the auxiliary trees first will become clear in the correctness proof given in §4.4. This is done to account for initial trees that may not have any non-ϵ terminal symbols in their frontiers. For example, there could be an initial tree of the form: $S\ ---\ S\ ---\ \epsilon$, where the second S node is an adjunction node. Also note that the data structure Γ is crucial to the algorithm. It enables us to perform the following operations efficiently: (1) update $LIST$s of nodes; (2) check if a given 4-tuple is in $LIST(\alpha)$ for some given α; etc.

4.3. An example. Before proving the correctness of TAL-PARSE, we demonstrate the basic algorithm above with an example. Consider the following language $\mathcal{L}$: $\{a^n b^n c^n | n \ge 0\}$. Figure 4 shows a TAG that generates $\mathcal{L}$. Here $*$ denotes the adjoin node. Integers ranging from 1 to 5 are used to label nodes and the label of each node is shown next to it. The input considered is *aabbcc*.

Stage 1: $LIST(4)$ is $\emptyset$. $LIST(3)$ is computed as $(4, i, i, 5)$ for every i and $(5, j, j, 6)$ for every j. $LIST(2) = (2, 3, 4, 5),\ (3, 4, 4, 5),\ (2, 3, 5, 6),\ (3, 4, 5, 6)$. Finally, $LIST(1)$ is computed as $(0, 1, 3, 5),\ (0, 2, 3, 5),\ (1, 3, 4, 5),\ (1, 3, 5, 6)$.

Stage 2: $LIST(4)$ still remains empty. $LIST(3)$ also remains the same. $LIST(2)$ gets modified because of the adjunction performed at this node. $LIST(2)$ gets a new element: $(0, 4, 4, 6)$. As a result, $LIST(1)$ also gets a new member: $(0, 4, 4, 6)$. In this case, TAL-PARSE returns "YES".

4.4. Correctness of TAL-PARSE. In this subsection, we prove the following lemma.

LEMMA 4.2. *The algorithm* TAL-PARSE *is correct.*

Proof. The proof will be by induction on ℓ. The execution of instructions 2–7 will constitute a "stage" of the algorithm. In the following proof, by a "terminal symbol," we mean a terminal symbol other than ϵ.

Induction hypothesis. After the ℓth stage of the algorithm, all tree nodes that span an input segment of length ℓ or less will have been identified. We say a node spans a string of

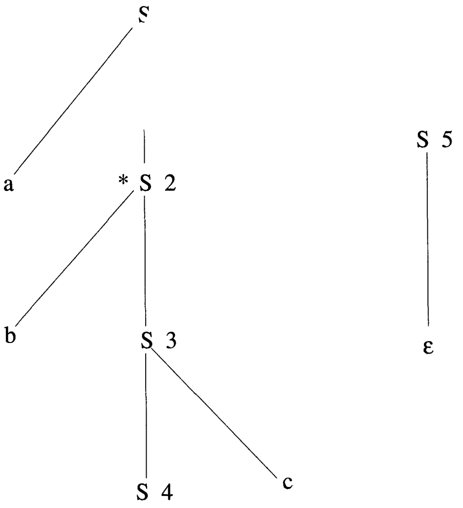


FIG. 4. *A TAG for* $\{a^n b^n c^n | n \geq 0\}$.

length ℓ if it spans $a_{i+1} \ldots a_j$ and $a_{k+1} \ldots a_m$ such that sum of lengths of these two segments is ℓ.

Base case ($\ell = 1$). If γ is any node that spans an input segment of length 1, then there are three possibilities: (1) γ is labeled by a terminal symbol; (2) the frontier of the subtree rooted at γ has a terminal symbol; or (3) γ has neither a terminal for a label nor a terminal in its frontier. Case (1) is trivial. In case (2), EVALUATE-TREE will surely infer that γ spans an input of length 1. In case (3), γ can span an input segment of length 1 only with the help of an adjoin operation. In particular, the auxiliary tree that is adjoined should have a span of length 1. But the roots of all the auxiliary trees fall under case (2) and hence would have been correctly processed in lines 2 and 3 of TAL-PARSE.

The induction step. Assume the hypothesis for all stages $\leq \ell$. We will prove it for the $(\ell + 1)$th stage.

Let γ be any tree node that spans an input segment of length $\ell + 1$. The corresponding derivation can be obtained with a sequence of basic operations, namely, composition and adjoin performed on trees whose span-lengths are ℓ or less. Some of these operations might not have added to the span-length of the derived tree. Call these operations useless and others useful. Let α be the node in this tree at which the last (from bottom-up) useful operation was performed. Now there are two cases to consider.

Case 1. The last useful operation performed was a composition—say, of two subtrees (of span-length at least one each). Clearly, both of these subtrees have a span-length of ℓ or less and hence, using the induction hypothesis, would have already been identified. Therefore, the procedure EVALUATE-TREE when performed on γ will identify the fact that γ spans an input segment of length $\ell + 1$.

Case 2. The last useful operation performed was adjoin—say, at a node α labeled X. This means that there is a quadruple—say, (i, j, k, m)—in $LIST(\alpha)$ such that when a derived auxiliary tree labeled X is adjoined at α, the resultant tree has a span-length of $\ell + 1$. Let δ be the derived tree corresponding to the quadruple (i, j, k, m) and θ be the auxiliary tree that is adjoined. In this case, if δ has at least one terminal in its frontier, then θ will have a span-length of at most ℓ. In θ, each constituent subtree will have a span-length of ℓ or less and hence would have been already identified. Thus the algorithm ADJOIN will identify α as having a span-length of $\ell + 1$.

On the other hand, if δ does not have any terminal in its frontier, then θ will have a span-length of $\ell + 1$. But auxiliary trees have at least one terminal in their frontiers and hence would have been processed (by TAL-PARSE) in lines 2 and 3 of stage $\ell + 1$. Therefore, in this case also, the procedure ADJOIN will identify α as having a span-length of $\ell + 1$. □

Time and space analysis. Since $LIST(\gamma)$ will have $O(n^4)$ quadruples for any node γ, EVALUATE-NODE(γ) takes time $O(n^6)$. In fact, EVALUATE-NODE can be run in $O(n^5)$ time. We only mention how to modify steps 6–11. (R is an array of lists indexed from 1 to n).

```
Scan through $LIST(\beta)$ and for each quadruple of the form
$(l, m, m, p)$, add $p$ to the list $R[l]$
for each $(i, j, k, l)$ in $LIST(\alpha)$ do
    for each $p \in R[l]$ do
        Put $(i, j, k, p)$ into $LIST(\gamma)$ if it is not there
```

ADJOIN takes time $O(n^6)$. Thus for any tree T, EVALUATE-TREE(T) also runs in time $O(n^6)$, implying that the run time of TAL-PARSE is $O(n^7)$. Clearly, the space used is $O(n^4)$. As a result, we get the following result.

THEOREM 4.3. *The algorithm* TAL-PARSE *takes* $O(n^7)$ *time and* $O(n^4)$ *space.*

4.5. Reducing the run time to $O(n^6)$. In §4.1, we reduced the time cost of our variant of the CKY CFL parser from $O(n^4)$ down to $O(n^3)$. In this subsection, we use the same technique for TALs also. That is, in stage q, we only generate trees of span-length exactly q.

We can process steps 6–11 of EVALUATE-NODE in $O(n^4)$ time as follows: Let R be an $n \times n$ matrix initialized to all zeros and Q be a list of tuples. In stage q of TAL-PARSE ($1 \leq q \leq n$), we generate only quadruples whose total span length is exactly q.

```
Scan through $LIST(\beta)$ and for each quadruple of the form
$(l, m, m, p)$ encountered, mark $R[l, p]$ and add $(l, p)$ to $Q$
for each $(i, j, k, l)$ in $LIST(\alpha)$ do
    Let $p = l + q - [(j - i) + (l - k)]$.
    if $R[l, p]$ is marked then
        Put $(i, j, k, p)$ into $LIST(\gamma)$ if it is not there
Using the list $Q$, clean the matrix $R$ for future use
```

Similarly, we could process ADJOIN in $O(n^5)$ time as follows. (Here α is the node into which the auxiliary tree rooted at β is adjoined. R is an $n^2 \times n^2$ matrix initialized to zeros.)

```
Scan through $LIST(\alpha)$ and for each quadruple of the form
$(m, j, k, r)$ encountered, mark $R[m, j, k, r]$
for each $(i, m, r, l)$ in $LIST(\beta)$ do
    Let $p = q - [(m - i) + (l - r)]$.
    for $v := 0$ to $p$ do
        if $R[m, m + v, r - p + v, r]$ is marked then
            Add $(i, m + v, r - p + v, l)$ into $LIST(\beta)$ if it is not there
Using the list $\beta$, clean the matrix $R$ for future use
```

Thus it becomes clear that EVALUATE-TREE runs in time $O(n^5)$, yielding the following result.

THEOREM 4.4. *The modified version of* TAL-PARSE *runs in time* $O(n^6)$ *using* $O(n^4)$ *space.*

Note. The worst-case and average-case run times of our algorithm are different. We believe that this algorithm will perform well in practice.

5. A faster algorithm for TAL parsing. In this section, we show that TAL parsing can be done in time $O(n\ M(n^2))$, where $M(k)$ is the time needed for multiplying two Boolean matrices of size $k \times k$ each. We make use of TAL-PARSE.

The claim follows from the following lemmas.

LEMMA 5.1. EVALUATE-NODE(γ) *can be processed in time* $n^2\ M(n)$ *for any node* γ.

Proof. It suffices to show how we could process lines 6–11 of EVALUATE-NODE(γ) within the stated time bound. Construct two Boolean matrices A and B corresponding to elements in $LIST(\alpha)$ and $LIST(\beta)$ as follows: The matrices will be of size $n^2 \times n^2$ each. Rows of the matrices are named with tuples of the form (i, j) and columns are numbered with tuples of the form (k, l) $(1 \le i, j, k, l \le n)$. $A[i, j; k, l]$ will be 1 if and only if (i, j, k, l) is a quadruple in $LIST(\alpha)$. Similarly, define B.

Clearly, (i, j, k, p) will be in $LIST(\gamma)$ if and only if

$$f_{ijkp} = \sum_{l=1}^{n}\sum_{m=1}^{n} A[i, j; k, l]\ B[l, m; m, p]$$

is a 1. However,

$$\sum_{l=1}^{n}\sum_{m=1}^{n} A[i, j; k, l]\ B[l, m; m, p] = \sum_{l} A[i, j; k, l]\left(\sum_{m} B[l, m; m, p]\right).$$

The above fact suggests the following strategy for processing EVALUATE-NODE(γ): Let V_{ik} be the $n \times n$ submatrix of A defined as

$$V_{ik} = \{A[i, j; k, l]\}, \quad 1 \le j, l \le n,$$

and let C be an $n \times n$ matrix such that $C[p, l] = \sum_{q=1}^{n} B[p, q; q, l]$, $1 \le p, l \le n$. Now, f_{ijkp} is simply the dot product of the jth row of V_{ik} and the pth column of C. Of course, the jth row in V_{ik} is nothing but all the elements $A[i, j; k, l]$, $1 \le l \le n$.

Therefore, multiplying the two $n \times n$ matrices V_{ik} and C yields the values of f_{ijkp} for $1 \le j, p \le n$. Hence obtaining values of all possible f_{ijkp}'s takes n^2 such multiplications. □

LEMMA 5.2. ADJOIN *can be performed in time* $M(n^2)$.

Proof. Say we have to adjoin at a node labeled α. Let the auxiliary tree to be adjoined be β. We have to generate all quadruples of the form (i, j, k, l) such that (i, q, r, l) is in $LIST(\beta)$ and (q, j, k, r) is in $LIST(\alpha)$. This can be done as follows. Generate matrices U and W for $LIST(\alpha)$ and $LIST(\beta)$, respectively, as above. Now the array U can be indexed (i.e., permuted) in such a way that the ilth row of U has the following elements:

$$U[i, 1; 1, l], U[i, 1; 2, l], \ldots, U[i, 1; n, l],\ U[i, 2; 1, l], U[i, 2; 2, l], \ldots, U[i, 2; n, l],$$
$$\ldots, U[i, n; n, l].$$

Likewise, index (i.e., permute) the elements of W such that the jkth column has the elements

$$W[1, j; k, 1], W[1, j; k, 2], \ldots, W[1, j; k, n], W[2, j; k, 1], W[2, j; k, 2], \ldots, W[2, j; k, n],$$
$$\ldots,\ W[n, j; k, n].$$

Clearly, the dot product and then Boolean OR of the ilth row of U and jkth column of W will be a 1 if and only if the adjoin results in the quadruple (i, j, k, l). □

As a result we get the following result (making use of TAL-PARSE and the above lemmas).

THEOREM 5.3. *The algorithm* TAL-PARSE *costs* $O(n\ M(n^2))$ *time and* $O(n^4)$ *space.*

Note. The above algorithm has a run time of $O(n^{5.752})$, which already breaks the $\Theta(n^6)$ barrier.

6. An $O(n^3M(n))$-time algorithm. We now show that TAL parsing can be done in time $O(n^3\ M(n))$. Since the best-known value for $M(k)$ is $O(k^{2.376})$, the time bound of our algorithm is $O(n^{5.376})$. The key idea is to use TAL-PARSE together with asymptotically fast algorithms for Boolean matrix multiplication. In stage q of the algorithm, we process only quadruples whose span length is exactly equal to q, $1 \leq q \leq n$. Boolean matrix multiplication has been previously employed in the design of parsing algorithms [18], [4] as has been mentioned before.

It should be noted here that it suffices to show how ADJOIN can be performed at any node in $O(n^2\ M(n))$ time in any given stage of the algorithm.

There can be only $O(n^3)$ quadruples of the form (i, j, k, l) whose span-length is exactly equal to q (for any q). These quadruples can be classified according to the value of $k - j$ as follows: Call a quadruple "a class-m quadruple" if $k - j = m$. Clearly, there are at most n such classes and there are at most n^2 quadruples in each class whose span length is q.

Consider any arbitrary class (say m) of quadruples. Let (i', j', k', l') be any quadruple in this class. Then the other quadruples (i, j, k, l) in this class can be enumerated as follows: $i = i' \pm c, l = l' \pm c, j = j' \pm d, k = k' \pm d$, for any integers c and d (as long as the tuple (i, j, k, l) is a meaningful one). That is, there are at most n choices for the pair i, l and, independently, there are at most n choices for the pair j, k.

Now construct two Boolean matrices J and K as follows: The ilth row of J will have entries corresponding to the quadruples

$$(i, 1, 1, l), (i, 1, 2, l), \ldots, (i, 1, n, l),\ (i, 2, 1, l), (i, 2, 2, l), \ldots, (i, 2, n, l),\ \ldots,\ (i, n, n, l).$$

Here il ranges over all possible values for the pair i, l indicated above. The entry corresponding to a quadruple (i, j, k, l) will be 1 if the node under consideration is known to span it. Similarly, the jkth column of K will have entries corresponding to the quadruples

$$(1, j, k, 1),(1, j, k, 2), \ldots,(1, j, k, n),(2, j, k, 1),(2, j, k, 2), \ldots,(2, j, k, n), \ldots,(n, j, k, n).$$

Here jk ranges over all possible values for the pair j, k indicated above.

Now, the product of the two matrices J and K will give information about all new quadruples in class m that arise out of adjunction at the node under concern. That is, a quadruple (i, j, k, l) in class m is spanned by the node under consideration only if the corresponding entry in the product is 1. Since J and K are matrices of size $n \times n^2$ and $n^2 \times n$, respectively, they can be multiplied in time $O(n\ M(n))$. Therefore, we can compute all the quadruples belonging to a specific class and spanned by the node under consideration in $O(n\ M(n))$ time. The technique employed for multiplication of J and K is as follows: Partition the matrix J into n submatrices $J_1, J_2, \ldots, J_n$, where J_i consists of the ith leftmost block of n columns of J. Similarly partition K into $K_1, K_2, \ldots, K_n$, where K_i is the ith topmost block of n rows of K, $1 \leq i \leq n$. Now, clearly, the blockwise product of J and K is the same as $\sum_{i=1}^{n} J_i\ K_i$. This implies that we can obtain all the quadruples spanned by this node (of a specific span length) in time $O(n^2\ M(n))$.

As a result, we have shown the following.

THEOREM 6.1. *TAL parsing can be done in time $O(n^3\ M(n))$ with space complexity $O(n^4)$.*

The following theorem pertains to parallel implementation of the above ideas. In TAL-PARSE, we can run everything in parallel fine, except we process one span-length at a time.

THEOREM 6.2. *TAL parsing can be done on the EREW PRAM in $O(n \log n)$ time using $(n^2M(n))/\log n$ processors.*

Proof. The theorem can be proven using the fact that two $n \times n$ matrices can be multiplied in $O(\log n)$ time on an EREW PRAM using $M(n)/\log n$ processors [7]. If the above algorithms are analyzed using this fact, the theorem readily follows. The PT (Processor × Time) bound of this algorithm is asymptotically the same as the sequential run time. Therefore, our parallel algorithm is an optimal implementation of the sequential algorithm. □

7. Conclusions. We have resolved the open question of whether TAL parsing can be done in time $o(n^6)$. Our algorithm also runs efficiently in parallel. We have presented algorithms for CFL and TAL parsing which may perform well in practice. Our parallel algorithm has time cost asymptotically within a log factor of any prior parallel algorithm for TAL parsing. At the same time, our algorithm uses significantly fewer processors. It is still unknown whether there is a *practical* algorithm for TAL parsing that runs in time $o(n^6)$.

Acknowledgments. I am grateful to the referees for their critical comments and suggestions. I am thankful to Günter Hotz and Aravind K. Joshi for their support and encouragement. I am also thankful to B. Srinivas and Shibu Yooseph for many stimulating discussions.

REFERENCES

[1] A. Aho and J. D. Ullman, *The Theory of Parsing, Translating, and Compiling*, vol. 1, Addison–Wesley, Reading, MA, 1984.

[2] D. Coppersmith and S. Winograd, *Matrix multiplication via arithmetic progressions*, in Proc. 19th Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1987, pp. 1–6; J. Symbolic Comput., 9 (1990), pp. 251–280.

[3] G. Gazdar, *Applicability of indexed grammars to natural languages*, Technical report CSLI-85-34, Center for Study of Language and Information, Stanford University, Stanford, CA, 1985.

[4] S. L. Graham, M. A. Harrison, and W. L. Ruzzo, *On line context free language recognition in less than cubic time*, in Proc. 8th Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1976, pp. 112–120.

[5] Y. Guan and G. Hotz, *An $O(n^5)$ recognition algorithm for coupled parenthesis rewriting systems*, in Proc. TAG+ Workshop, University of Pennsylvania Press, Philadelphia, 1992.

[6] K. Harbusch, *An efficient parsing algorithm for tree adjoining grammars*, in Proc. 28th Meeting of the Association for Computational Linguistics, Morgan Kaufmann, San Francisco, 1990, pp. 284–291.

[7] J. Já Já, *An Introduction to Parallel Algorithms*, Addison–Wesley, Reading, MA, 1992, p. 248.

[8] A. K. Joshi, *How much context-sensitivity is necessary for characterizing structural descriptions: Tree adjoining grammars*, in Natural Language Processing: Theoretical, Computational and Psychological Perspective, D. Dowty, L. Karttunen, and A. Zwicky, eds., Cambridge University Press, New York, 1985, pp. 112–133.

[9] A. K. Joshi, L. S. Levy, and M. Takahashi, *Tree adjunct grammars*, J. Comput. System Sci., 10 (1975), pp. 136–163.

[10] A. Kroch and A. K. Joshi, *Linguistic relevance of tree adjoining grammars*, Technical Report MS-CIS-85-18, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, 1985.

[11] T. Nurkkala and V. Kumar, *A parallel parsing algorithm for natural language using tree adjoining grammar*, in Proc. 8th International Parallel Processing Symposium, IEEE Computer Society Press, Los Alamitos, CA, 1994, pp. 315–320.

[12] M. Palis, S. Shende, and D. S. L. Wei, *An optimal linear time parallel parser for tree adjoining languages*, SIAM J. Comput., 19 (1990), pp. 1–31.

[13] B. H. Partee, A. Ter Meulen, and R. E. Wall, *Studies in Linguistics and Philosophy*, vol. 30, Kluwer Academic Publishers, Norwell, MA, 1990.

[14] G. Satta, *Tree adjoining grammar parsing and Boolean matrix multiplication*, in Proc. 32nd Meeting of the Association for Computational Linguistics, Morgan Kaufmann, San Francisco, 1994.

[15] G. Satta, personal communication, September 1993.

[16] Y. Schabes and A. K. Joshi, *An Earley-type parsing algorithm for tree adjoining grammars*, in Proc. 26th Meeting of the Association for Computational Linguistics, Morgan Kaufmann, San Francisco, 1988, pp. 258–269.

[17] M. Steedman, *Dependency and coordination in the grammar of Dutch and English*, Language, 61 (1985), pp. 523–568.

[18] L. G. Valiant, *General context-free recognition in less than cubic time*, J. Comput. System Sci., 10 (1975), pp. 308–315.

[19] K. Vijayashanker and A. K. Joshi, *Some computational properties of tree adjoining grammars*, in Proc. 23rd Meeting of the Association for Computational Linguistics, Morgam Kaufmann, San Francisco, 1985, pp. 82–93.

AN EFFICIENT PARALLEL ALGORITHM FOR THE MATRIX-CHAIN-PRODUCT PROBLEM*

PRAKASH RAMANAN†

Abstract. We consider the problem of finding an optimal order of computing a matrix-chain product. This problem can be solved using dynamic programming in $O(n^3)$ sequential time, but the best sequential algorithm known for this problem runs in $O(n \log n)$ time. A general technique for parallelizing a class of dynamic-programming algorithms leads to an algorithm that runs in $O(\log^2 n)$ time on a concurrent-read exclusive-write parallel random-access machine (CREW PRAM) with $O(n^6/\log n)$ processors. The best parallel algorithm previously known runs in $O(\log^3 n)$ time using $O(n^2/\log^3 n)$ processors. We present an algorithm that runs in $O(\log^4 n)$ time on a CREW PRAM with n processors.

Key words. matrix-chain product, polygon triangulation, dynamic programming, parallel algorithm, PRAM model, processor-time complexity

AMS subject classifications. 68Q10, 68Q20, 68Q22, 68Q25

1. Introduction.

Dynamic programming is a widely used technique for solving various problems in computer science and operations research. Many such problems reduce to computing the quantity $c(1, n)$ based on a recurrence of the following type: for $1 \le i < j \le n$,

$$c(i, j) = \begin{cases} g(i) & \text{if } j = i + 1, \\ \min\limits_{i<k<j} c(i, k) + c(k, j) + f(i, k, j) & \text{if } j > i + 1, \end{cases}$$

where $g(i)$, $1 \le i < n$, and $f(i, k, j)$, $1 \le i < k < j \le n$, are known in advance. We are required to compute $c(1, n)$ and an optimal value of k at each of the intermediate steps.

Dynamic-programming algorithms for these problems can be implemented by sequential straight-line programs of size $O(n^3)$ using the operations min and + (see [1]). Guibas et al. [7] presented a general procedure for implementing such algorithms on very large-scale integration (VLSI). Here we consider parallel algorithms in the more powerful PRAM (parallel random-access machine) model. The general method of Valiant et al. [18] for the parallelization of sequential straight-line programs leads to $O(\log^2 n)$-time algorithms using $O(n^9)$ processors. Using the specific features of this class of dynamic-programming algorithms, Rytter [15] presented $O(\log^2 n)$-time algorithms on the CREW (concurrent-read exclusive-write) PRAM with $O(n^6/\log n)$ processors. Rytter demonstrated his method on three typical problems of this class: finding an optimal order of computing a matrix-chain product, constructing an optimal binary search tree, and finding an optimal triangulation of a polygon (see [1]).

In this paper we concentrate on the problem of finding an optimal order of computing a matrix-chain product. This matrix-chain-product problem (MCPP) can be stated as follows.

MCPP. Given positive integers $(w_1, w_2, \ldots, w_n)$, find an optimal order of computing the matrix chain product $M_1 \times M_2 \times \cdots \times M_{n-1}$, where M_i is of dimensions $w_i \times w_{i+1}$ and the cost of computing the product of a $w_i \times w_j$ matrix with a $w_j \times w_k$ matrix is $w_i w_j w_k$.

MCPP can be formulated as a problem of the type discussed above. In the corresponding recurrence relation, we have $g(i) = 0$, $1 \le i < n$, and $f(i, k, j) = w_i w_j w_k$.

Hu and Shing [8] showed that MCPP is equivalent to a problem of finding a minimum-weight triangulation of a convex polygon with weighted vertices. A convex polygon is specified by a list (i.e., a sequence) of vertices, in cyclic order, around the boundary of the polygon. Let

*Received by the editors February 8, 1993; accepted for publication (in revised form) January 4, 1995. This research was supported in part by a 1992 Summer Research Fellowship from the Liberal Arts and Sciences College of Wichita State University.

†Department of Computer Science, Wichita State University, Wichita, KS 67260-0083 (ramanan@cs.twsu.edu).

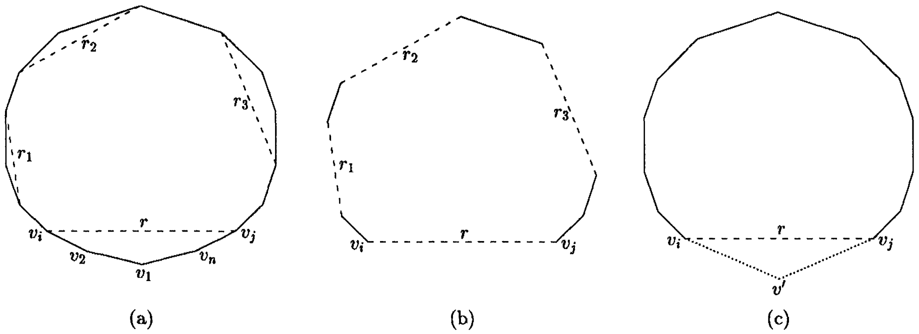


FIG. 1.

$P = (v_1, v_2, \dots, v_n)$ be a convex polygon (see Figure 1a). An *edge* $v_i v_{i \bmod n+1}$, $1 \le i \le n$, is a straight line segment that connects the two adjacent vertices v_i and $v_{i \bmod n+1}$ of the polygon. An *arc* $v_i v_j$ is a straight line segment that connects the two nonadjacent vertices v_i and v_j of the polygon. Each vertex v_i, $1 \le i \le n$, has a *weight* $w_i > 0$ associated with it. A *triangulation* (of the interior) of P consists of $n-2$ triangles formed by the n edges and $n-3$ nonintersecting arcs. The *cost* of a triangle $v_i v_j v_k$ is $w_i w_j w_k$. The *cost* of a triangulation T is the sum of costs of all the triangles in T. The triangulation problem (TP) we are interested in is as follows.

TP. Given $(w_1, w_2, \dots, w_n)$, find a minimum-cost triangulation of P.

Hu and Shing [8, 9] showed that MCPP is equivalent to TP; the transformation between the two problems is straightforward and can be carried out in constant time on a CREW PRAM with n processors. They also presented an $O(n \log n)$ sequential algorithm for TP. Ramanan [14] gave a simpler presentation of their algorithm. This algorithm, which is quite sequential, is described in §2. Our parallel algorithm is based on a divide-and-conquer version of this algorithm. In §3, we describe the basic ideas behind our divide-and-conquer approach. In §§4 and 5, we present the divide-and-conquer algorithm and a parallel implementation of it. The resulting algorithm runs in $O(\log^4 n)$ time on a CREW PRAM with n processors. In comparison, Bradford [4] and Czumaj [6] presented $O(\log^3 n)$-time algorithms using $O(n^3/\log n)$ and $O(n^2/\log^3 n)$ processors, respectively.

2. The sequential algorithm. Let $P = (v_1, v_2, \dots, v_n)$ be a convex polygon (see Figure 1a). For the sake of simplicity, we assume that no two vertices of P have the same weight. Also, without loss of generality, we let v_1 be the vertex of smallest weight in P; such a vertex is said to be a *global minimum*. A *global maximum* vertex is defined analogously. A vertex v_i is said to be a *local minimum* if $w_i < \min(w_{i-1}, w_{i+1})$. A *local maximum* vertex is defined analogously. Note that the number of local minimum vertices and local maximum vertices are the same. P is said to be *m-modal* if it has m local minimum vertices.

Let $v_{n+1} \equiv v_1$ and $w_{n+1} \equiv w_1$. Let $i, j, 1 \le i < j-1 \le n, v_i \ne v_j$, be such that $w_k > \max(w_i, w_j)$ for all $k, i < k < j$. Then $r = v_i v_j$ is called a *horizontal arc* (*h-arc*) and $P(r) \equiv (v_i, v_{i+1}, \dots, v_j)$ is called the *upper subpolygon* of P that is *bounded below* by r. In our figures, dashed lines are used to represent h-arcs (see Figure 1). Hu and Shing [8] showed that there are $n-3$ h-arcs, and that no two of them intersect; so they constitute a triangulation of P. They also presented a linear-time algorithm for finding all the h-arcs.

We let $w(v)$ denote the weight of a vertex v. For an h-arc r, we let r^1 and r^2 denote the two end-points of r such that $w(r^1) < w(r^2)$; r^1 and r^2 will be referred to as the *lower* and *upper* endpoints of r. Let r, r_1, and r_2 be h-arcs of P. r_1 is said to be *below* r_2 (or r_2 is *above* r_1), denoted by $r_1 \le r_2$ (or $r_2 \ge r_1$), if all the vertices of $P(r_2)$ are also vertices of $P(r_1)$. Two h-arcs are said to be *comparable* if one of them is below the other; otherwise, they are said to be *incomparable.* r_1 is said to be *properly below* r_2 (or r_2 is *properly above* r_1), denoted by $r_1 < r_2$ (or $r_2 > r_1$), if $r_1 \leq r_2$ and $r_1 \neq r_2$. For $r_1 \leq r_2$, r is said to be *between* r_1 and r_2 if $r_1 \leq r \leq r_2$; r is said to be *properly between* r_1 and r_2 if $r_1 < r < r_2$.

A *subpolygon* of P is a polygon each of whose edges is either an edge or an h-arc of P. Note that the h-arcs of a subpolygon of P are those h-arcs of P that are in the subpolygon. If r_i, $1 \leq i \leq k$, are pairwise incomparable h-arcs, we let $P(;r_1, r_2, \ldots, r_k)$ denote the subpolygon obtained from P as follows: For each i, $1 \leq i \leq k$, remove from P all the vertices of $P(r_i)$ except r_i^1 and r_i^2. This subpolygon is said to be *bounded above* by r_i, $1 \leq i \leq k$. If r is another h-arc such that $r < r_i$, $1 \leq i \leq k$, we let $P(r; r_1, r_2, \ldots, r_k)$ denote the subpolygon obtained from $P(r)$ in a similar manner, i.e., $P(r; r_1, r_2, \ldots, r_k) = (P(r))(; r_1, r_2, \ldots, r_k)$ (see Figure 1b). This subpolygon is said to be *bounded below* by r and *bounded above* by r_i, $1 \leq i \leq k$.

A *fan* of a (sub)polygon is a triangulation of the (sub)polygon in which the vertex u of smallest weight is connected to all the other vertices; u is called the *center* of the fan. For a subpolygon S of P, we let $F(S)$ denote the fan of S; $C_F(S)$ denotes its cost. $F(r; r_1, r_2, \ldots, r_k)$ denotes the fan of $P(r; r_1, r_2, \ldots, r_k)$; $C_F(r; r_1, r_2, \ldots, r_k)$ denotes its cost. The following is from Hu and Shing [9, Lem. 1].

LEMMA 2.1 (see [9]). *Let H be the set of h-arcs in an optimal triangulation of P. If a subpolygon of P that is bounded by some of the h-arcs in H contains none of these h-arcs in its interior, then a fan is an optimal triangulation of the subpolygon.*

By Lemma 2.1, we only need to determine which of the $n - 3$ h-arcs are in an optimal triangulation of P. For this, we need to be able to efficiently compute the cost of a fan of a subpolygon. Let S be a subpolygon of P that is bounded below by an h-arc r (and possibly bounded above by some h-arcs). The *weight* of S is defined as $W(S) = \sum w_a w_b$, where the summation is taken over all the edges $v_a v_b$ of S except r. We let $W'(S)$ denote $W(S) - w(r^1)w(v)$, where v is the neighbor of r^1 other than r^2 in S. Then $C_F(S) = w(r^1)W'(S)$. The *weight* of r is $W(r) \equiv W(P(r))$; $W(r; r_1, r_2, \ldots, r_k)$ denotes $W(P(r; r_1, r_2, \ldots, r_k))$. Note that

$$W(r; r_1, r_2, \ldots, r_k) = W(r) - \sum_{i=1}^{k}[W(r_i) - w(r_i^1)w(r_i^2)].$$

$C_F(r; r_1, r_2, \ldots, r_k) = w(r^1)W'(r; r_1, r_2, \ldots, r_k)$. The modality, the local minima and maxima, and the weights of all the h-arcs can be easily computed in linear time. Then, for any subpolygon S, $C_F(S)$ can be easily computed using the weights of the h-arcs that bound S.

The next concept we introduce is the *cutoff value* of an h-arc. For a subpolygon S of P that is bounded below by an h-arc r, we let $S'(r)$ denote the polygon obtained from $S(r) = S$ by inserting a new (special) vertex v' between r^1 and r^2. For example, for $r = v_i v_j$, $i < j$, $P'(r) = (v', v_i, v_{i+1}, \ldots, v_j)$ (see Figure 1c). The cutoff value of r, denoted by $co(r)$, is defined to be *the* value of $w(v')$ for which the following holds: The minimum cost of a triangulation of $P'(r)$ that contains r equals the minimum cost of a triangulation of $P'(r)$ that does not contain r; i.e., there exists an optimal triangulation of $P'(r)$ that contains r, and there exists another optimal triangulation of $P'(r)$ that does not contain r. In [14], we proved the following.

LEMMA 2.2 (see [14]). *For an h-arc r, $co(r)$ exists and is unique*; *also, $co(r) < w(r^1)$. Moreover, if $w(v') < co(r)$, then no optimal triangulation of $P'(r)$ can contain r*; *if $w(v') > co(r)$, then every optimal triangulation of $P'(r)$ will contain r*; *if $w(v') = co(r)$, then there exists an optimal triangulation of $P'(r)$ that contains r, and there exists another optimal triangulation of $P'(r)$ that does not contain r.*

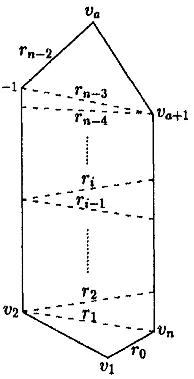


FIG. 2.

For $w(v') < w(r^1)$, we let $F'(r; r_1, r_2, \dots, r_k)$ denote the fan of $P'(r; r_1, r_2, \dots, r_k)$ (centered at v'); its cost is $w(v')W(r; r_1, r_2, \dots, r_k)$.

Note that once the cutoff values of all the h-arcs in P have been found, we can determine the h-arcs in an optimal triangulation of P by performing a bottom-up scan in linear time. In §2.1, we describe a linear-time algorithm for finding an optimal triangulation of unimodal polygons. In §2.2, we describe its extension to an $O(n \log n)$ algorithm for general multimodal polygons.

2.1. Algorithm for unimodal polygons. Let $P = (v_1, v_2, \dots, v_n)$ be an unimodal polygon, where v_1 is the global minimum and v_a is the global maximum (see Figure 2). Henceforth, in our figures, we let the relative order of the y-coordinates of the vertices of an unimodal (sub)polygon be the same as the relative order of their weights. Any two h-arcs of P are comparable; therefore, the $n-3$ h-arcs of P are one above the other. Let the h-arcs be labeled $r_1, r_2, \dots, r_{n-3}$ from bottom to top. Let $l \in \{2, n\}$ be such that $w_l = \min(w_2, w_n)$; the *bottom edge* $r_0 = v_1 v_l$ and the *top edge* $r_{n-2} = r_{n-3}^2 v_a$ are considered to be *degenerate* h-arcs. For $1 \le i \le n-2$, r_i and r_{i-1} share the end point r_i^1; i.e., r_i^1 is either r_{i-1}^1 or r_{i-1}^2.

We now describe a linear-time algorithm for finding an optimal triangulation of P. The algorithm performs a scan from top to bottom and processes the h-arcs one by one in the order $r_{n-2}, r_{n-3}, \dots, r_1, r_0$. When it processes r_i, it constructs an optimal triangulation $T(r_i)$ of $P(r_i)$, and computes its cost $C(r_i)$, and $co(r_i)$. By Lemma 2.1, $T(r_i)$ can be represented by a list $L1(r_i)$ of the h-arcs that are in $T(r_i)$; $L1(r_i)$ contains these h-arcs in bottom to top order, and its first element is r_i. Note that the cutoff values of these arcs need not be in decreasing order. After processing r_i, the algorithm has the two lists $L1(r_i)$ and $L2(r_i)$. $L2(r_i)$ is defined as follows: It is a sublist (i.e., a subsequence) of $L1(r_i)$; its first element is r_i; an element of $L1(r_i)$ is in $L2(r_i)$ iff its cutoff value is less than that of all the preceding elements in $L1(r_i)$.

Now, we show how to update $L1$ and $L2$ when the algorithm processes r_{i-1}. $L1(r_{i-1})$ is obtained from $L1(r_i)$ as follows: Remove the longest prefix of elements (i.e., h-arcs) all of which have cutoff values greater than or equal to $w(r_{i-1}^1)$, and then insert r_{i-1} at the front. Let r_j be the second element of $L1(r_{i-1})$. Then $T(r_{i-1})$ consists of $T(r_j)$ and $F(r_{i-1}; r_j)$; its cost is $C(r_{i-1}) = C(r_j) + C_F(r_{i-1}; r_j)$.

Before we can get $L2(r_{i-1})$, we need to compute $co(r_{i-1})$. We take $co(r_{n-2})$ to be 0.

$$\begin{aligned} co(r_{n-3}) &= w(r_{n-3}^1)w(r_{n-3}^2)w_a/[w_a(w(r_{n-3}^1) + w(r_{n-3}^2)) - w(r_{n-3}^1)w(r_{n-3}^2)] \\ &= w_{a-1}w_a w_{a+1}/[w_a(w_{a-1} + w_{a+1}) - w_{a-1}w_{a+1}]. \end{aligned}$$

Before we can compute $co(r_{n-4})$, we need to know whether or not r_{n-3} will get "cutoff" at $co(r_{n-4})$, i.e., whether $co(r_{n-4}) \le co(r_{n-3})$ or $co(r_{n-4}) > co(r_{n-3})$.

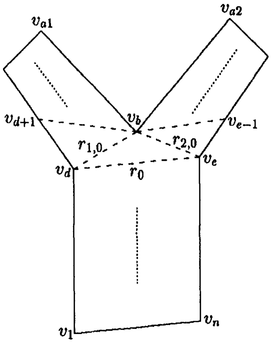


FIG. 3.

We need the following notation. Let L be a list of h-arcs in decreasing order of cutoff values, and let the last element in L have cutoff value zero. Let r be any h-arc. *Locating* $co(r)$ with respect to L means the following: Find $co(r)$ to be greater than the cutoff value of the first element in L, equal to the cutoff value of a particular element in L, or properly between the cutoff values of a particular pair of adjacent elements in L.

For $i - 1 \leq n - 4$, $co(r_{i-1})$ is computed by solving an equation. To set up this equation, we need to know which h-arcs will exist in an optimal triangulation of $P'(r_{i-1})$ when $w(v')$ is less than but arbitrarily close to $co(r_{i-1})$. This is determined by locating $co(r_{i-1})$ with respect to $L2(r_i)$, as explained in [14]. Finally, $L2(r_{i-1})$ is obtained from $L2(r_i)$ as follows: Remove the longest prefix of elements all of which have cutoff values greater than or equal to $co(r_{i-1})$, and then insert r_{i-1} at the front.

$L1(r_0)$ is the list of h-arcs in an optimal triangulation $T(r_0)$ of $P(r_0) = P$, and $C(r_0)$ is the cost of $T(r_0)$. The above algorithm can compute $C(r_{i-1})$, $co(r_{i-1})$, and $L2(r_{i-1})$ from $L2(r_i)$ without using $L1(r_i)$. So we can compute $co(r_i)$ and $L2(r_i)$ for i varying from $n - 3$ down to 0 without keeping track of $L1$.

2.2. Algorithm for multimodal polygons.

In this subsection, we describe the $O(n \log n)$ algorithm for general multimodal polygons. First, we describe the algorithm for bimodal polygons. Let $P = (v_1, v_2, \ldots, v_n)$ be a bimodal polygon, where v_1 is the global minimum, v_b is a local minimum, and v_{a1} and v_{a2} are the local maxima, $1 < a1 < b < a2 < n$; $w_n < w_b$ (see Figure 3). Then there exists a vertex v_d, $1 \leq d < a1$, such that $w_d < w_b < w_{d+1}$; also, there exists a vertex v_e, $a2 < e \leq n$, such that $w_e < w_b < w_{e-1}$. $r_{1,0} = v_d v_b$, $r_{2,0} = v_b v_e$ and $r_0 = v_d v_e$ are h-arcs of P. $P(r_{1,0})$, $P(r_{2,0})$, and $P(; r_0)$ are unimodal polygons. The h-arcs of $P(; r_0)$ will be labeled as $r_1, r_2, \ldots$ from top to bottom. Without loss of generality, let $w_d < w_e$.

As in §2.1, we let $T(r)$ denote an optimal triangulation of $P(r)$ with cost $C(r)$. The algorithm for computing the cutoff values for the above bimodal polygon P works as follows: First find the lists $L2(r_{1,0})$ and $L2(r_{2,0})$ for unimodal subpolygons $P(r_{1,0})$ and $P(r_{2,0})$, respectively, using the algorithm in §2.1. Since $w_d < w_e$, let $L2''(r_{2,0})$ be the list obtained from $L2(r_{2,0})$ by removing the longest prefix of elements all of which have cutoff values greater than or equal to w_d. If $r_{2,i}$ is the first element of $L2''(r_{2,0})$, then $T(r_0)$ consists of $T(r_{1,0})$, $T(r_{2,i})$ and $F(r_0; r_{1,0}, r_{2,i})$; its cost is

$$C(r_0) = C(r_{1,0}) + C(r_{2,i}) + w_d W(r_{2,0}; r_{2,i}).$$

Let $L2'(r_0)$ be the list obtained by *merging* $L2(r_{1,0})$ and $L2''(r_{2,0})$ into a single list such that the cutoff values of the elements are in decreasing order from the front. Using $L2'(r_0)$, we can compute $co(r_0)$ and obtain $L2(r_0)$ as explained in §2.1. Then we can perform a top to bottom scan of $P(;r_0)$; for $i \geq 1$, $co(r_i)$ and $L2(r_i)$ can be computed from $L2(r_{i-1})$ as explained in §2.1. This completes our description of the algorithm for bimodal polygons.

In the above bimodal polygon, if $r_{1,i}$ and $r_{2,j}$ are h-arcs of $P(r_{1,0})$ and $P(r_{2,0})$, respectively, then they are incomparable; but they are above r_0. The h-arc r_0 is called a *bridge*. In general, a *bridge* is an h-arc r whose endpoints are the lower endpoints of two other h-arcs r' and r''; r' and r'' will have the same upper endpoint. The preceding paragraph describes a general procedure for obtaining $L2(r)$ from $L2(r')$ and $L2(r'')$. In general, an m-modal polygon will have $m-1$ bridges. The above top-down algorithm for bimodal polygons can be easily extended to an $O(n \log n)$ algorithm for general m-modal polygons (see [14]).

3. Basics of the divide-and-conquer approach.

Let $P = (v_1, v_2, \ldots, v_n)$ be a general polygon. Our parallel algorithm, described in §§4 and 5, is based on a divide-and-conquer version of the algorithm presented in §2. In this section, we describe the basic ideas behind the divide-and-conquer approach and prove some required results. First, we need the following two lemmas.

LEMMA 3.1. *Let T_1 be an optimal triangulation of P that contains the minimum number of h-arcs, and let T_2 be any other optimal triangulation of P. Then all the h-arcs in T_1 are also in T_2. So, there exists a unique optimal triangulation of P that contains the minimum number of h-arcs.*

Proof. Consider the h-arcs in T_1 from the bottom up. Let r_1 be an h-arc that is in T_1 but not in T_2, and suppose that all the h-arcs properly below r_1 in T_1 are also in T_2. Let r_2 be the topmost h-arc below r_1 that is in T_2 (r_2 could be the bottom edge, which is a degenerate h-arc). By Lemma 2.2, since T_2 does not contain r_1 or any other h-arc properly between r_2 and r_1, we have that $co(r_1) \geq w(r_2^1)$. But then r_1 can be removed from T_1, giving an optimal triangulation with one less h-arc. This is a contradiction. So, all the h-arcs in T_1 are also in T_2. □

For a subpolygon S of P, let $C(S)$ denote the cost of an optimal triangulation of S.

LEMMA 3.2. *For an h-arc r' in P, $co(r') = \max_S C(S)/[W(S) - w((r')^1)w((r')^2)]$, where the maximum is over all the subpolygons S of P that are bounded below by r'.*

Proof. At $w(v') = co(r')$, let $T_1'(r')$ be an optimal triangulation of $P'(r')$ that contains r'; let $T_2'(r')$ be the triangulation that contains the minimum number of h-arcs among all optimal triangulations of $P'(r')$ that do not contain r'; their costs are equal. There exists a subpolygon $S = P(r'; r_1, r_2, \ldots, r_k)$ such that $T_2'(r')$ contains r_i, $1 \leq i \leq k$, and $F'(r'; r_1, r_2, \ldots, r_k)$ (see Figure 1). By Lemma 2.2, since $T_2'(r')$ contains the minimum number of h-arcs, we have $co(r_i) < w(v') = co(r') < w((r')^1)$, for $1 \leq i \leq k$. So, $T_1'(r')$ also will contain r_i, $1 \leq i \leq k$. Since both $T_1'(r')$ and $T_2'(r')$ are optimal, the corresponding triangulations in $S'(r')$ must have the same cost; i.e., $co(r')W(S) = C(S) + co(r')w((r')^1)w((r')^2)$. Then the result follows from Lemma 2.2. □

To describe our divide-and-conquer approach, we need the following notations. For a subpolygon S of P, let $T(S)$ denote an optimal triangulation of S and let $C(S)$ be its cost; if S is bounded by some of the h-arcs in an optimal triangulation T of P, we can take $T(S)$ to be the optimal triangulation of S that is contained in T. If $S = P(r'; r_1, r_2, \ldots, r_k)$, we let $T(r'; r_1, r_2, \ldots, r_k) \equiv T(S)$, and $C(r'; r_1, r_2, \ldots, r_k) \equiv C(S)$. For an h-arc r' in S, let $co(S, r')$ denote the cutoff value of r' in S.

Let P be split along one of the h-arcs r into $P(1) = P(r)$ and $P(2) = P(;r)$. For $j = 1, 2$, we let $P(j, r'; r_1, r_2, \ldots, r_k) \equiv (P(j))(r'; r_1, r_2, \ldots, r_k)$. Let $T(j)$ denote an optimal triangulation of $P(j)$, with cost $C(j)$. For a subpolygon S that is bounded by some of the h-arcs in $T(j)$, let $T(j, S)$ denote the optimal triangulation of S that is contained in $T(j)$ with cost $C(j, S) = C(S)$; if $S = P(j, r'; r_1, r_2, \ldots, r_k)$, we let $T(j, r'; r_1, r_2, \ldots, r_k) \equiv T(j, S)$ and $C(j, r'; r_1, r_2, \ldots, r_k) \equiv C(j, S)$. Let $co(j, r')$ denote the cutoff value of r' in $P(j)$.

As before, $T(r')$ denotes an optimal triangulation of $P(r')$ with cost $C(r')$, and $co(r')$ is the cutoff value of r' in P.

Our divide-and-conquer approach is as follows: For $j = 1, 2$, separately compute $T(j)$; as byproducts, we also get $T(j, r')$, $C(j, r')$, and $co(j, r')$ for the h-arcs r' in $T(j)$. We need to *merge* the results for $j = 1, 2$ to obtain $T(r')$, $C(r')$, and $co(r')$ for the h-arcs r' in an optimal triangulation T of P. For r' in $T(1)$, since $P(1, r') \equiv P(r')$, we have $T(r') = T(1, r')$, $C(r') = C(1, r')$, and $co(r') = co(1, r')$. To obtain $T(r')$, $C(r')$, and $co(r')$ for the other h-arcs r' in T, we need the following results.

LEMMA 3.3. *For an h-arc $r2$ in $P(2)$, $co(r2) \geq co(2, r2)$.*

Proof. Follows from Lemma 3.2, since any subpolygon S of $P(2)$ is also a subpolygon of P. □

LEMMA 3.4. *Let T be the optimal triangulation of P that contains the minimum number of h-arcs. Any h-arc in T is also in $T(1)$ or $T(2)$.*

Proof. First, we show that any h-arc of $P(1)$ in T will also be in $T(1)$. Let $r1$ be an h-arc of $P(1)$ in T such that no h-arc of $P(1)$ properly below $r1$ is in T. Then, by Lemma 2.2, we have $co(1, r1) = co(r1) < w(r^1)$. So $r1$ and, by Lemma 3.1, all the h-arcs above $r1$ in T will also be in $T(1)$. Hence any h-arc of $P(1)$ in T will also be in $T(1)$.

Now we show that any h-arc of $P(2)$ in T will also be in $T(2)$. Consider the h-arcs of $P(2)$ in T from the bottom up. Let $r2$ be an h-arc of $P(2)$ in T such that all the h-arcs of $P(2)$ properly below $r2$ that are in T are also in $T(2)$. Let $r2_1$ and $r2_2$ be the topmost h-arcs properly below $r2$ that are in T and $T(2)$, respectively. Since $r2_1 \leq r2_2$, we have $w(r2_1^1) \leq w(r2_2^1)$. Since $r2$ is in T, we have $co(r2) < w(r2_1^1)$. By Lemma 3.3, we have $co(2, r2) \leq co(r2)$. Putting the last three inequalities together, we have $co(2, r2) < w(r2_2^1)$. So, by Lemma 2.2, $r2$ will also be in $T(2)$. Hence any h-arc of $P(2)$ in T will also be in $T(2)$. □

From now onwards, our model of computation will be a CREW PRAM with n processors. In the following sections, using the above results, we present a divide-and-conquer version of the algorithm in §2 and a parallel implementation of it. Sections 4 and 5 contain $O(\log^2 n)$ and $O(\log^4 n)$ algorithms for unimodal and multimodal polygons, respectively. Czumaj [6] also presented an $O(\log^2 n)$ algorithm for unimodal polygons.

4. Parallel algorithm for unimodal polygons. In this section, we present an $O(\log^2 n)$ algorithm for unimodal polygons. Let P be an unimodal polygon with n vertices. First, assign one processor to each vertex. By comparing the weight of each vertex with those of its two neighbors, we can find the global minimum vertex and the global maximum vertex in constant time. Then, in constant time, we can relabel the vertices with respect to the global minimum vertex. So, let $P = (v_1, v_2, \ldots, v_n)$ as described in §2.1, where v_1 is the global minimum, and v_a is the global maximum (see Figure 2). Since $(w_1, w_2, \ldots, w_a)$ and $(w_n, w_{n-1}, \ldots, w_{a+1})$ are in increasing order, we can merge the two lists into a single sorted list in $O(\log \log n)$ time (see [10, 17]). From this sorted list, we can obtain the $n - 3$ h-arcs, and label them as $r_1, r_2, \ldots, r_{n-3}$ from bottom to top, as in §2.1, in $O(\log n)$ time. Now assign one processor to each h-arc. We can compute the weights of all the h-arcs by performing a *parallel prefix computation* (see [10, 11]), from top to bottom, in $O(\log n)$ time.

We need the following definitions. Let r' be an element of a list L. A *proper predecessor* of r' in L is an element that precedes r' in L. A *predecessor* of r' in L is either r' or a proper predecessor of r'. The *immediate predecessor* of r' in L is the element that immediately precedes r' in L. *Proper successor*, *successor*, and *immediate successor* are defined analogously.

Our divide-and-conquer algorithm and its parallel implementation is as follows: Assign one processor to each h-arc of P. Divide P along the middle h-arc $r = r_{(n-3) \text{ div } 2}$ into $P(1) = P(r)$ and $P(2) = P(;r)$. For $j = 1, 2$, let $L1(j)$ and $L2(j)$ be the $L1$ and $L2$ lists

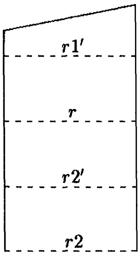


FIG. 4.

corresponding to an optimal triangulation $T(j)$ of $P(j)$ as described in §2.1. Note that r will be the last element of $L1(2)$ and $L2(2)$; it will also be the first element of $L1(1)$ and $L2(1)$. Compute in parallel for $j = 1, 2$, $L1(j)$ and $L2(j)$; as byproducts, we also get $T(j, r')$, $C(j, r')$, and $co(j, r')$ for $r' \in L1(j)$. Let $L1(j, r')$ denote the list of successors of r' in $L1(j)$; it is also the list of h-arcs in $T(j, r')$.

Let $L1$ and $L2$ be the lists corresponding to an optimal triangulation T of P. We show how to *merge* the results for $P(1)$ and $P(2)$ to obtain $L1$ and $L2$ in $O(\log n)$ time. For $r1 \in L1(1)$, we have $T(r1) = T(1, r1)$, $C(r1) = C(1, r1)$, and $co(r1) = co(1, r1)$. We need to find $T(r2)$, $C(r2)$, and $co(r2)$ for $r2 \in L1(2)$. To find $T(r2)$, we need the following lemma.

LEMMA 4.1. *Let $r2 \in L1(2)$. There exist $r2' \in L1(2, r2)$ and $r1' \in L2(1)$ such that the following hold* (*see Figure* 4) :

(i) *$r1'$ is the first element* (*from the bottom*) *in $L2(1)$ such that $co(r1') < w((r2')^1)$.*

(ii) *The triangulation of $P(r2)$ that consists of $T(2, r2; r2')$, $T(r1')$, and $F(r2'; r1')$ is an optimal triangulation $T(r2)$ of $P(r2)$.*

Proof. Let $T_{\min}(r2)$ be the optimal triangulation of $P(r2)$ that contains the minimum number of h-arcs. Let $r1'$ be the bottommost h-arc of $P(1)$ in $T_{\min}(r2)$, and let $r2'$ be the top most h-arc of $P(2)$ in $T_{\min}(r2)$. Then, by Lemma 3.4, $r1' \in L1(1)$ and $r2' \in L1(2, r2)$. Also, since $T_{\min}(r2)$ contains the minimum number of h-arcs, by Lemma 2.2, $r1' \in L2(1)$ and (i) must hold.

$T_{\min}(r2)$ consists of $T_{\min}(r2; r2')$, $T_{\min}(r1')$, and $F(r2'; r1')$. $T(r)$ and $T(2, r2)$ are optimal triangulations of $P(1)$ and $P(2, r2)$, respectively, and contain $r1'$ and $r2'$, respectively. So, the triangulation $T(r2)$ as described in (ii) has the same cost as $T_{\min}(r2)$ and hence is optimal. □

COROLLARY 4.2. *Let $r2 \in L1(2)$. For $r2' \in L1(2, r2)$, let $r1'$ be the first element* (*from the bottom*) *in $L2(1)$ such that $co(r1') < w((r2')^1)$* (*see Figure* 4). *Let*

$$savings(r2') = C(2, r2') + C(r; r1') - C_F(r2'; r1')$$

(*$savings(r)$ is taken to be* 0). *Pick $r2' \in L1(2, r2)$ such that $savings(r2')$ is maximized. Then the triangulation of $P(r2)$ that consists of $T(2, r2; r2')$, $T(r1')$, and $F(r2'; r1')$ is an optimal triangulation of $P(r2)$.*

Proof. Let $T_1(r2)$ be the triangulation of $P(r2)$ that consists of $T(2, r2)$ concatenated with $T(r)$; its cost is $C_1(r2) = C(2, r2) + C(r)$. For any $r2' \in L1(2, r2)$, let $r1'$ be as specified in the corollary. Let $T_2(r2)$ be the triangulation of $P(r2)$ specified in the corollary. It differs from $T_1(r2)$ only in $P(r2'; r1')$. Its cost is

$$\begin{aligned} C_2(r2) &= C(2, r2; r2') + C(r1') + C_F(r2'; r1') \\ &= C_1(r2) - savings(r2'). \end{aligned}$$

By Lemma 4.1, $T_2(r2)$ must be an optimal triangulation of $P(r2)$ for some $r2' \in L1(2, r2)$. Clearly, this $r2'$ must be such that it maximizes $C_1(r2) - C_2(r2) = savings(r2')$. □

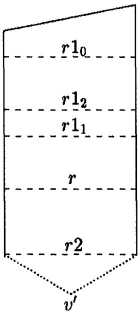


FIG. 5.

In the above corollary, if $r2' = r1' = r$, then $T(r2)$ consists of $T(2, r2)$ and $T(r)$.

Now we show how to obtain $T(r2)$ and $C(r2)$, for each $r2 \in L1(2)$, in $O(\log n)$ time. For each $r2' \in L1(2)$, the processor assigned to $r2'$, using binary search in $O(\log n)$ time, finds the first element $r1'$ in $L2(1)$ such that $co(r1') < w((r2')^1)$. Then the processor computes $savings(r2')$ in constant time. Once all the processors assigned to the h-arcs in $L1(2)$ have computed their *savings*, they perform a prefix computation from top to bottom, in $O(\log n)$ time, using the max operation, to find the largest *savings*. Among all the h-arcs in $L1(2, r2)$, let $r2'$ have the largest *savings*; since $savings(r) = 0$, we have $savings(r2') \geq 0$. By Corollary 4.2, $T(r2)$ consists of $T(2, r2; r2')$, $T(r1')$, and $F(r2'; r1')$; its cost is $C(r2) = C(2, r2) + C(r) - savings(r2')$. Thus $C(r2)$ can be obtained for all $r2 \in L1(2)$ in $O(\log n)$ time. When $r2$ is the bottom degenerate h-arc of P, we get an optimal triangulation T of P and the corresponding list $L1$ in $O(\log n)$ time.

Now, we show how to compute $co(r2)$ for each $r2 \in L1(2)$ in $O(\log n)$ time. Before we can compute $co(r2)$, we need to know which h-arcs will exist in an optimal triangulation of $P'(r2)$ when $w(v')$ is less than but arbitrarily close to $co(r2)$ (see Figure 5). Let $T_1'(r2)$ be the triangulation of $P'(r2)$ that consists of $T(r2)$ and the triangle $v'r2^1r2^2$. For any value of $w(v')$, $T_1'(r2)$ is of minimum cost among all triangulations of $P'(r2)$ that contain $r2$. The processor assigned to $r2$ can compute its cost in constant time as $C_1'(r2) = C(r2) + w(v')w(r2^1)w(r2^2)$. For any given value of $w(v')$, let $T_2'(r2)$ be a triangulation that is of minimum cost among all triangulations of $P'(r2)$ that do not contain $r2$; let $C_2'(r2)$ be its cost. By Lemma 2.2, we have the following: for $w(v') < co(r2)$, $C_2'(r2) < C_1'(r2)$; at $w(v') = co(r2)$, $C_2'(r2) = C_1'(r2)$.

By Lemma 3.3, we have $co(r2) \geq co(2, r2)$. To determine if $co(r2) = co(2, r2)$ or $co(r2) > co(2, r2)$, we need the following two lemmas.

LEMMA 4.3. *For any $w(v') \leq co(r2)$, if $T_2'(r2)$ contains any h-arc below r, then $co(r2) = co(2, r2)$.*

Proof. If $T_2'(r2)$ contains any h-arc below r, then $co(r2)$ is independent of $P(r)$; so $co(r2) = co(2, r2)$. □

LEMMA 4.4. *Let $r1_0$ be the first element in $L2(1)$ such that $co(r1_0) < co(2, r2)$ (see Figure 5). Let $T_0'(r2)$ be the triangulation of $P'(r2)$ that consists of $T(r1_0)$ and $F'(r2; r1_0)$; let $C_0'(r2)$ be its cost. Then $co(r2) > co(2, r2)$ iff $C_0'(r2) < C_1'(r2)$ at $w(v') = co(2, r2)$.*

Proof. Let $w(v') = co(2, r2) \leq co(r2)$. By Lemma 2.2, $T_0'(r2)$ is of minimum cost among all triangulations of $P'(r2)$ that do not contain any h-arc properly below r; we have $C_0'(r2) \geq C_2'(r2)$. If $co(r2) = co(2, r2)$, then $C_0'(r2) \geq C_2'(r2) = C_1'(r2)$. If $co(r2) > co(2, r2)$, then, by Lemma 4.3, $T_2'(r2)$ can not contain any h-arc below r; then $T_2'(r2)$ can be taken to be $T_0'(r2)$; so $C_0'(r2) = C_2'(r2) < C_1'(r2)$. □

The processor assigned to $r2$, using binary search, in $O(\log n)$ time, finds $r1_0$ as specified in the above lemma. Then the processor computes $C_0'(r2)$ (at $w(v') = co(2, r2)$), in constant time, as $C'_0(r2) = C(r1_0) + co(2, r2)W(r2; r1_0)$. If $C'_0(r2) \geq C'_1(r2)$, the processor sets $co(r2) = co(2, r2)$. Now consider the case $C'_0(r2) < C'_1(r2)$; we have $r1_0 > r$ and $co(r2) > co(2, r2)$. $co(r2)$ lies between the cutoff values of an adjacent pair of predecessors of $r1_0$ in $L2(1)$. Also, by Lemma 4.3, when $w(v') \leq co(r2)$, $T'_2(r2)$ can not contain any h-arc below r. The processor assigned to $r2$ can locate $co(r2)$ with respect to $L2(1)$ as follows: The processor, in a binary search manner, picks a proper predecessor $r1_1$ of $r1_0$ in $L2(1)$ with $co(r1_1) < w(r2^1)$ (see Figure 5). (If no such $r1_1$ exists, then $co(r2)$ lies between the cutoff values of $r1_0$ and its immediate predecessor in $L2(1)$). Then the processor *tentatively* assigns the value of $co(r1_1)$ to $w(v')$. Let $r1_2$ be the immediate successor of $r1_1$ in $L2(1)$. Let $T'_3(r2)$ be the triangulation of $P'(r2)$ that consists of $T(r1_2)$ and $F'(r2; r1_2)$. For $co(r1_2) < w(v') \leq co(r1_1)$, $T'_3(r2)$ is of minimum cost among all triangulations of $P'(r2)$ that do not contain any h-arc below r. The processor assigned to $r2$ computes its cost at $w(v') = co(r1_1)$, in constant time, as

$$C'_3(r2) = C(r1_2) + co(r1_1)W(r2; r1_2).$$

By Lemmas 2.2 and 4.3, we have $co(r2) < w(v') = co(r1_1)$ iff $C'_1(r2) < C'_3(r2)$; also, $co(r2) = w(v') = co(r1_1)$ iff $C'_1(r2) = C'_3(r2)$. Thus, by performing binary search in $L2(1)$, the processor can locate $co(r2)$ to be either equal to the cutoff value of some predecessor of $r1_0$ or properly between the cutoff values of an adjacent pair of predecessors of $r1_0$. In the latter case, $co(r2)$ can be obtained by solving an equation. So, $co(r2)$ can be computed in $O(\log n)$ time.

Final $L2$ list can be obtained in $O(\log n)$ time as follows. By performing a parallel prefix computation in $L1$, find the smallest cutoff value $co(r'')$ preceding each $r' \in L1$. Then $r' \in L2$ iff $co(r') < co(r'')$.

We have shown that the merge operation can be performed in $O(\log n)$ time. So, we have the following.

THEOREM 4.5. *An optimal triangulation of an unimodal polygon can be found in $O(\log^2 n)$ time.*

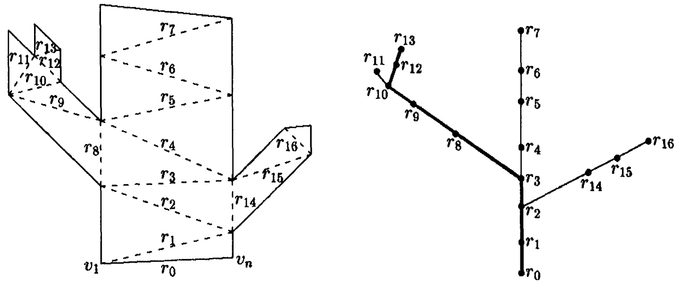
5. Parallel algorithm for multimodal polygons. Let P be a general multimodal polygon with n vertices. In this section, we present an $O(\log^4 n)$ algorithm for finding an optimal triangulation of P. The description of the algorithm is divided into the following subsections. The names of the variables used in a subsection are local to that subsection.

5.1. Finding the local minima, maxima, modality, and the h-arcs. First, assign one processor to each vertex of P. We can find the global minimum vertex by performing a parallel prefix computation in $O(\log n)$ time. Then, in constant time, we can relabel the vertices with respect to the global minimum vertex. So let $P = (v_1, v_2, \ldots, v_n)$, where v_1 is the global minimum. By comparing the weight of each vertex with those of its two neighbors, we can find the local minima and maxima in constant time. Then we can find the modality m (i.e., the number of local minima) by performing a parallel prefix computation in $O(\log n)$ time.

Now we show how to find all the h-arcs in $O(\log n)$ time. Let $w_{n+1} \equiv w_1$. Suppose that $r = v_i v_j$, $i < j$, is an h-arc. If $w_i < w_j$, then i must be the largest index, $1 \leq i < j$, such that $w_i < w_j$; if $w_i > w_j$, then j must be the smallest index, $i < j \leq n + 1$, such that $w_j < w_i$. So, as pointed out in [4, 6], finding all the h-arcs of P is equivalent to the following all-nearest-smaller-value-pairs problem (ANSVP) [3].

ANSVP. Let $(w_1, w_2, \ldots, w_n)$ be a sequence of real numbers, where w_1 is the smallest; let $w_{n+1} \equiv w_1$. For each j, $1 < j \leq n$, find the largest index i, $1 \leq i < j$, such that $w_i < w_j$; also, find the smallest index k, $j < k \leq n + 1$, such that $w_k < w_j$.

Berkman et al. [3] showed that this problem can be solved in $O(\log n)$ time. So, all the h-arcs can be found in $O(\log n)$ time.



(a) (b)

FIG. 6.

5.2. Finding the trunk and the weights. As shown in [9], P corresponds to a rooted binary tree, where each h-arc corresponds to a node (see Figure 6). A node (i.e., h-arc) r' is the parent of another node r'' if r'' is *immediately above* r' (i.e., r'' is properly above r' and there is no h-arc properly between them). Let $l \in \{2, n\}$ be such that $w_l = \min(w_2, w_n)$. The degenerate h-arc $v_1 v_l$ is the root and is at the bottom. Parent-to-child edges are directed upwards. Only those h-arcs that are *bridges* (see §2.2) have two children. After finding the h-arcs, we can set up the tree in constant time. Then, using the *Euler tour technique* [10, 16], in $O(\log n)$ time, we can compute the sizes (i.e., the number of nodes) of the subtrees rooted at each of the nodes. Using the same technique, we can also compute the weights of all the h-arcs in $O(\log n)$ time.

Throughout the rest of this paper, the term *vertex* will refer to a vertex of the polygon P; unless specified otherwise, the term *node* will refer to a node of the tree (i.e., an h-arc of P) defined in the previous paragraph.

The *trunk* of the tree is a linear chain of nodes defined as follows: The root is on the trunk; if a non-bridge node is on the trunk, then its child is on the trunk; for a bridge node on the trunk, its child with the larger subtree is on the trunk. In Figure 6b, the trunk is shown in dark lines. Assigning one processor to each tree node, the trunk can be found in $O(\log n)$ time.

5.3. Outline of the divide-and-conquer algorithm. Our divide-and-conquer algorithm for finding an optimal triangulation of P is as follows. Assigning one processor to each node of P, find its trunk $\hat{T}$. A *basic subtree* (equivalently, *basic subpolygon*) is a subtree (i.e., subpolygon) that results when the trunk is removed from the tree. For a basic subpolygon S, its *base* is its bottom edge; i.e., the node that is adjacent to a (bridge) node on the trunk. Note that for any h-arc r in S, we have $S(r) = P(r)$. For each basic subpolygon S, recursively find the following:

(a) the set $set1(S)$ of h-arcs that are in an optimal triangulation of S; this is analogous to the $L1$ list described in §2.1, but it is a set instead of being a list because S might not be unimodal;

(b) $C(r)$ and $co(r)$ for each $r \in set1(S)$. $T(r)$ will contain those h-arcs of $P(r)$ that are in $set1(S)$;

(c) the list $L2(S)$ for the subpolygon S, analogous to the $L2$ list described in §2.2;

(d) for each $r \in L2(S)$, the *cumulative cost* $CC(S, r)$ and the *cumulative weight* $CW(S, r)$ defined in the next subsection.

Now we need to *glue* the subtrees to the trunk. This means obtaining the above four items for the whole polygon P.

The trunk $\hat{T}$ of P can be found in $O(\log n)$ time. By our choice of trunk, each basic subtree will have at most $n/2$ nodes. We will show that the glue operation can be performed in

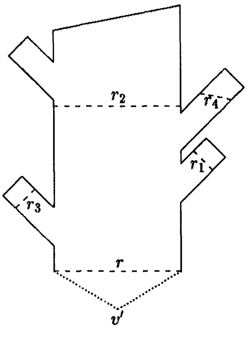


FIG. 7.

$O(\log^3 n)$ time. So, the run-time for the divide-and-conquer part of the algorithm is $DC(n) \leq DC(n/2) + O(\log^3 n) = O(\log^4 n)$. As shown in §§5.1 and 5.2, all the other operations can be performed in $O(\log n)$ time. So, the total run-time is $O(\log^4 n)$.

5.4. The cumulative cost and weight. Let $S = P(r)$ for an h-arc r. For some given $w(v') < w(r^1)$, we want to find an optimal triangulation $T'(r)$ of $P'(r)$ (see Figure 7). Suppose that, for some $r_1 \in L2(S)$, $w(v') > co(r_1)$, but $w(v')$ is no more than the cutoff values of all the proper predecessors of r_1 in $L2(S)$. Then, by Lemma 2.2, there exist pairwise incomparable h-arcs $r_i \in L2(S)$, $1 \leq i \leq k$, such that $T'(r)$ consists of r_i and $T(r_i)$, $1 \leq i \leq k$, and $F'(r; r_1, r_2, \ldots, r_k)$. We define the *cumulative set* of r_1 with respect to S as $cset(S, r_1) = \{r_1, r_2, \ldots, r_k\}$. The cost of $T'(r)$ is

$$C'(r) = \sum_{i=1}^{k} C(r_i) + w(v') \left[W(r) - \sum_{i=1}^{k} (W(r_i) - w(r_i^1)w(r_i^2)) \right].$$

We define the *cumulative cost* and *cumulative weight*, respectively, as follows: $CC(S, r_1) = \sum_{i=1}^{k} C(r_i)$; $CW(S, r_1) = \sum_{i=1}^{k} [W(r_i) - w(r_i^1)w(r_i^2)]$. Then

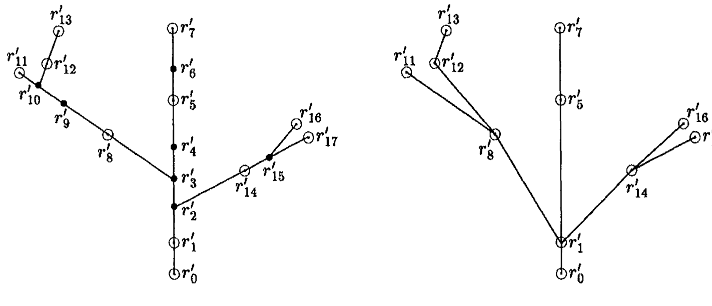
$$C'(r) = CC(S, r_1) + w(v')[W(r) - CW(S, r_1)].$$

Given any $w(v')$, we want to be able to quickly compute $C'(r)$. So, we want to efficiently precompute $CC(S, r_1)$ and $CW(S, r_1)$, for all $r_1 \in L2(S)$. First note that $cset(S, r_1)$ is the set obtained using the following procedure.

```
$L(r_1) \leftarrow$ the list obtained from $L2(S)$ by deleting all the proper predecessors of $r_1$
$cset(S, r_1) \leftarrow \{r_1\}$
From $L(r_1)$, delete $r_1$ and all the elements that are above $r_1$
while $L(r_1) \neq \phi$ do
    Let $r'$ be the first element of $L(r_1)$
    $cset(S, r_1) \leftarrow cset(S, r_1) \cup \{r'\}$
    From $L(r_1)$, delete $r'$ and all the elements that are above $r'$
```

So, $CC(S, r_1)$ and $CW(S, r_1)$ can be obtained by summing up $C(r')$ and $W(r') - w((r')^1)w((r')^2)$, respectively, over *some* of the successors of r_1 in $L2(S)$ (namely, those successors that are in $cset(S, r_1)$). We would like to split up $C(r')$ and $W(r') - w((r')^1)w((r')^2)$ into disjoint parts over the successors of r_1 in $L2(S)$. Then $CC(S, r_1)$ and $CW(S, r_1)$ can be obtained by summing up these parts over *all* the successors of r_1 in $L2(S)$. These parts will be called the *differential cost* and *differential weight*, respectively.

Unlike the cumulative cost and weight, the differential cost and weight of an h-arc $r' \in L2(S)$ depend only on $S(r') = P(r')$. Let *compact*(S) be the tree consisting only of



(a) (b)

FIG. 8.

the nodes (i.e., h-arcs) in $L2(S)$, such that the ancestor–descendant relationship among the nodes in $compact(S)$ is same as that in S. $compact(S)$ is obtained from S by deleting the nodes not in $L2(S)$, while preserving the ancestor-descendant relationship; in general, it will not be binary. Consider the tree S in Figure 8a, where the nodes in $L2(S)$ are circled; Figure 8b shows $compact(S)$. Given $L2(S)$, $compact(S)$ can be obtained in $O(\log n)$ time as follows:

```
$compact(S) \leftarrow S$
for $t = 1$ to $\log n$ do
    for all $r' \in compact(S)$ do in parallel
        if $parent(r') \neq nil$ and $parent(r') \notin L2(S)$ then $parent(r') \leftarrow parent(parent(r'))$
(* Comment. Now $parent(r') \in L2(S)$ for all $r' \in L2(S)$ *)
for all $r'$ in $compact(S)$ do in parallel
    if $r' \notin L2(S)$ then delete $r'$ from $compact(S)$
```

The root of $compact(S)$ is the root r of S. For $r' \in L2(S)$, we define the *differential set* $dset(r')$ to be the set of children of r' in $compact(S)$. Let $dset(r') = \{r'_1, r'_2, \dots, r'_p\}$ for some p (see Figure 7 with all the r s replaced by r' s). We define the *differential cost* to be $DC(r') = C(r') - \sum_{i=1}^{p} C(r'_i)$ and the *differential weight* to be

$$DW(r') = W(r') - w((r')^1)w((r')^2) - \sum_{i=1}^{p}[W(r'_i) - w(r'^1_i)w(r'^2_i)].$$

By assigning $|dset(r')|$ processors to r', we can compute $DC(r')$ and $DW(r')$, for all $r' \in L2(S)$, in $O(\log n)$ time.

For $r'' \in L2(S)$, $DC(r'')$ and $DW(r'')$ depend only on some subpolygon of $P(r'')$, and the subpolygons corresponding to different r'''s are disjoint. Also, it follows by induction that $\sum_{r''} DC(r'') = C(r')$ and $\sum_{r''} DW(r'') = W(r') - w((r')^1)w((r')^2)$, where the summation is over all the descendants r'' of r' (including r') in $compact(S)$. It follows that $\sum_{r''} DC(r'') = CC(S, r_1)$ and $\sum_{r''} DW(r'') = CW(S, r_1)$, where the summation is over all the successors r'' of r_1 (including r_1) in $L2(S)$.

Suppose that we have obtained $L2(S)$. We can obtain $compact(S)$ in $O(\log n)$ time. Then we can compute $DC(r')$ and $DW(r')$ for each $r' \in L2(S)$ in $O(\log n)$ time. Then, by performing a parallel prefix computation in $L2(S)$ from the rear (i.e., in increasing order of cutoff values), we can sum up the differential costs and weights (separately) to obtain $CC(S, r_1)$ and $CW(S, r_1)$ for each $r_1 \in L2(S)$ in $O(\log n)$ time.

5.5. The glue operation. Now we show how to *glue* the basic subtrees to the trunk. This will not alter the cutoff values of the h-arcs in the basic subtrees; it only involves computing the cutoff values of the h-arcs in the trunk and obtaining the overall set *set*1 and list $L2$ for P.

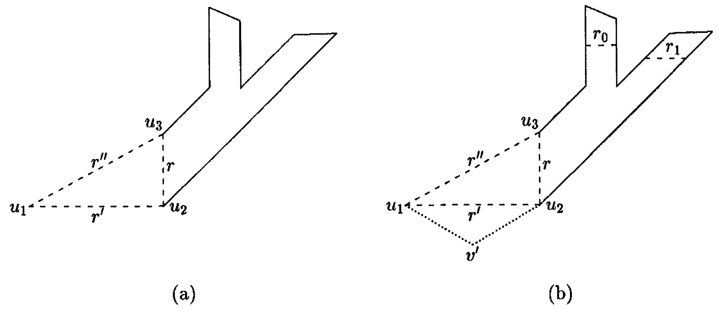


FIG. 9.

First, we glue each basic subtree to its adjacent bridge node on the trunk. Let S be a basic subtree with base r (so $S = P(r)$). Let r' be the parent of r in P; r' is a bridge node that lies on the trunk of P (see Figure 9a). Let r'' be the other child of r' in P; r'' is also on the trunk of P. Let r' have endpoints u_1 and u_2 and r'' have endpoints u_1 and u_3; then $r^1 = u_2$ and $r^2 = u_3$. We want to first glue S to r' and r'' to obtain the results for $P(r'; r'')$. This only involves finding $T(r'; r'')$, $C(r'; r'')$, and $co(P(r'; r''), r')$; $co(P(r'; r''), r'')$ is zero.

$T(r'; r'')$ and $C(r'; r'')$ are obtained as follows: The processor assigned to r', using binary search, in $O(\log n)$ time, finds the first entry r_1 in $L2(S)$ such that $co(r_1) < w(u_1)$. Then $T(r'; r'')$ is the triangulation $T'(r)$ described at the beginning of the previous subsection with v' replaced by u_1; its cost is

$$C(r'; r'') = CC(S, r_1) + w(u_1)[W(r) - CW(S, r_1)].$$

So, $T(r'; r'')$ and $C(r'; r'')$ can be found in $O(\log n)$ time.

Now we show how to compute $co(P(r'; r''), r')$ (see Figure 9b). To set up the equation for $co(P(r'; r''), r')$, we have to locate it with respect to $L2(S)$. This is done as follows: Let $T_1'(r'; r'')$ be the triangulation of $P'(r'; r'')$ that consists of r', $T(r'; r'')$, and the triangle $v'u_1u_2$. For any value of $w(v')$, $T_1'(r'; r'')$ is of minimum cost among all triangulations of $P'(r'; r'')$ that contain r'. The processor assigned to r' computes its cost as $C_1'(r'; r'') = C(r'; r'') + w(v')w(u_1)w(u_2)$. Then the processor, in a binary search manner, picks a candidate $r_0 \in L2(S)$ with $co(r_0) < \min(w(u_1), w(u_2))$ and *tentatively* assigns the value of $co(r_0)$ to $w(v')$; let r_1 be the immediate successor of r_0 in $L2(S)$. Let $T'(r)$ be the optimal triangulation of $P'(r)$ described in the previous subsection at $w(v') = co(r_0)$. Its cost is

$$C'(r) = CC(S, r_1) + co(r_0)[W(r) - CW(S, r_1)].$$

Let $T_2'(r'; r'')$ be the triangulation of $P'(r'; r'')$ that consists of $T'(r)$ and the triangle $v'u_1u_3$. For $co(r_1) < w(v') \leq co(r_0)$, $T_2'(r'; r'')$ is of minimum cost among all triangulations of $P'(r'; r'')$ that do not contain r'. The processor assigned to r' computes its cost at $w(v') = co(r_0)$ as

$$C_2'(r'; r'') = C'(r) + co(r_0)w(u_1)w(u_3).$$

By Lemma 2.2, we have $co(P(r'; r''), r') < w(v') = co(r_0)$ iff $C_1'(r'; r'') < C_2'(r'; r'')$; also, $co(P(r'; r''), r') = w(v') = co(r_0)$ iff $C_1'(r'; r'') = C_2'(r'; r'')$. Thus, by performing binary search on $L2(S)$, the processor can locate $co(P(r'; r''), r')$ to be equal to the cutoff value of some element in $L2(S)$, properly between the cutoff values of two adjacent elements in $L2(S)$, or greater than $co(r)$. In the latter two cases, $co(P(r'; r''), r')$ can be obtained by solving an equation. So, $co(P(r'; r''), r')$ can be computed in $O(\log n)$ time.

We have shown that a basic subtree can be glued to its adjacent bridge node on the trunk in $O(\log n)$ time. Now consider the general problem of gluing many basic subtrees to a long trunk. We first glue each basic subtree to its adjacent bridge node as explained above and then *combine* the results together. We will show that the combine operation can be performed in $O(\log^3 n)$ time; so, the glue operation can be performed in $O(\log^3 n)$ time.

5.6. The combine operation. Let $\hat{T}$ be the trunk of P. Assume that each basic subtree has been glued to its adjacent bridge node, as explained in the previous subsection. The *combine* operation combines all the results to obtain the final cutoff values of the h-arcs in the trunk and the overall results for P.

The combine operation is performed in a divide-and-conquer manner. Split the trunk $\hat{T}$ into two at the middle h-arc r; let $\hat{T}(1)$ and $\hat{T}(2)$ be the top and bottom halves of $\hat{T}$. For $j = 1, 2$, let $P(j)$ be the tree that consists of $\hat{T}(j)$ and all the basic subtrees glued to it; so, $P(1) = P(r)$ and $P(2) = P(; r)$. For $j = 1, 2$, recursively find the following:

(a) the set $set1(j)$ of h-arcs that are in an optimal triangulation of $P(j)$; the list $\hat{L}1(j)$ (in bottom to top order) of those h-arcs in $\hat{T}(j)$ that are in $set1(j)$;

(b) $C(j, r')$ and $co(j, r')$ for each $r' \in \hat{L}1(j)$; $T(j, r')$ will contain those h-arcs of $P(j, r')$ that are in $set1(j)$;

(c) the list $L2(j)$ for $P(j)$, analogous to the $L2$ list described in §2.2; the sublist $\hat{L}2(j)$ of $L2(j)$ consisting of those h-arcs that are also in $\hat{L}1(j)$;

(d) $CC(j, r')$ and $CW(j, r')$ (with respect to $P(j)$) for each $r' \in L2(j)$.

Now we need to *merge* the results for $P(1)$ and $P(2)$; this means obtaining the above four items for the whole polygon P. We will show that the merge operation can be performed in $O(\log^2 n)$ time; so, the combine operation can be performed in $O(\log^3 n)$ time.

5.7. The range tree in general. Our merge operation uses the *range tree* data structure of Lueker [12] and Willard [19] (also see [13]). It is typically used for representing a set of points in the (x, y)-plane. It consists of a primary structure and several secondary structures. The primary structure is the *segment tree* of Bentley [2] (also see [13]). For positive integers i and j, $i \leq j$, let the interval $[i, j] \equiv [i, j + 1) \equiv (i - 1, j]$ denote $\{i, i + 1, \ldots, j\}$. The segment tree on $[i, j]$ is a rooted binary tree defined as follows: If $i = j$, it consists of a single leaf. If $i < j$, its root corresponds to the interval $[i, j]$; the left and right subtrees are segment trees on $[i, \lfloor(i + j)/2\rfloor]$ and $(\lfloor(i + j)/2\rfloor, j]$, respectively.

The segment tree ST on $[1, n]$ has height $\lceil\log n\rceil$, and it can be constructed in $O(n)$ sequential time. For a node u in ST, let $Int(u)$ denote the interval corresponding to u. For an arbitrary given interval $[i, j] \subseteq [1, n]$, a node u in ST is called an *allocation node* if $Int(u) \subseteq [i, j]$, but $Int(parent(u)) \not\subseteq [i, j]$. The intervals corresponding to the allocation nodes for $[i, j]$ are disjoint, and their union equals $[i, j]$. There are at most $2\lceil\log n\rceil - 2$ allocation nodes for any $[i, j]$, and they can be found in $O(\log n)$ sequential time (see [13]).

Now we are ready to describe the range tree. Let S be a set of points in the (x, y)-plane. Let the x-coordinates of the points be in $[1, n]$, where $|S| = O(n)$. The y-coordinates of the points are real numbers; so, the y-range $[y_1, y_2]$ will denote the range of real y-values $y_1 \leq y \leq y_2$. For a range tree RT on S, the primary structure is the segment tree ST on the x-range $[1, n]$. For a node u in ST, let S_u be the subset of S consisting of those points whose x-coordinates are in $Int(u)$. The secondary structure associated with u is an array A_u that consists of the points in S_u in decreasing order of y-coordinates. Each point in A_u has two pointers (called *threads*) associated with it; see [13] for details.

The storage requirement for RT is $O(n \log n)$, and it can be constructed in $O(n \log n)$ sequential time. Let $R = [x_1, x_2] \times [y_1, y_2]$ be an orthogonal rectangle; i.e., R spans the x-range $[x_1, x_2] \subseteq [1, n]$ and the y-range $[y_1, y_2]$; R is referred to as a *range query*. We can obtain the points of S contained in R, in $O(\log n)$ sequential time, as follows: First, find the allocation nodes for the x-range $[x_1, x_2]$ in ST. Let A be the secondary structure (i.e., array) associated with the root of RT. Using binary search (for y_2 and y_1), locate the topmost

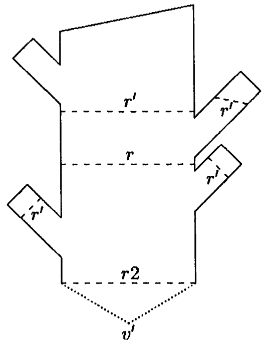


FIG. 10.

and bottommost points p_T and p_B in A with y-coordinates in the range $[y_1, y_2]$. For each allocation node u, we can locate the topmost and bottommost points of A_u that are contained in R by following the threads from p_T and p_B and going down the segment tree to the allocation nodes.

5.8. Our range tree. Let $r2 \in \hat{L}1(2)$. For a given $w(v') < w(r2^1)$, let $T''(r2)$ denote a triangulation that is of minimum cost among all triangulations of $P'(r2)$ that do not contain any h-arc properly below r (see Figure 10); let $C''(r2)$ be its cost. During the merge operation, we need to be able to efficiently compute $T''(r2)$ and $C''(r2)$ for different values of $w(v')$. For this, we use the range tree data structure.

We need the following definitions and notations. For two h-arcs r_1 and r_2 in P, $r_1 \geq r_2$, the *distance* between r_1 and r_2, denoted by $dist(r_1, r_2)$, is defined as follows: If $r_1 = r_2$, then $dist(r_1, r_2) = 0$; if $r_1 > r_2$, then $dist(r_1, r_2)$ is one plus the number of h-arcs that are properly between r_1 and r_2.

Let S be a basic subpolygon glued to $\hat{T}(2)$; let $r2'$ be the bridge node in $\hat{T}(2)$ that is adjacent to the base of S in P. For any h-arc r' in S, and any h-arc $r2'' \geq r2'$ in $\hat{T}(2)$, we define the *trunk distance* $tdist(r2'', r')$ to be $dist(r2'', r2')$. Recall that r is the topmost node of $\hat{T}(2)$. We let $tdist(r')$ to be the short form for $tdist(r, r')$.

Let $set2(2)$ be the (set) union of the $L2$ lists of all the basic subtrees glued to $\hat{T}(2)$. Each $r' \in set2(2)$ can be considered as a point in the (x, y)-plane; its x-coordinate is $tdist(r')$; its y-coordinate is $co(r')$. We set up the range tree RT on $set2(2)$. The x-range corresponding to the root of RT is $[1, n]$. The y-coordinates (i.e., cutoff values) are all nonnegative. The h-arcs in $set2(2)$ can be sorted according to their cutoff values in $O(\log n)$ time using the parallel merge-sort algorithm of Cole [5], and the range tree RT can be set up in $O(\log^2 n)$ time.

We augment the secondary structures in RT with some additional information related to cumulative costs and weights. Consider a node u in RT. Let $set2_u(2)$ be the subset of $set2(2)$ consisting of those h-arcs whose x-coordinates are in $Int(u)$. The secondary structure (i.e., array) A_u contains these h-arcs in decreasing order of their cutoff values. For any r_1 in A_u, let the *range cumulative set* $cset(u, r_1)$ be the set $cset(S, r_1)$ obtained using the pseudocode in §5.4 when $L2(S)$ is replaced by A_u. Let $CC(u, r_1)$ and $CW(u, r_1)$ be the *range cumulative cost* and *weight* obtained from $cset\ (u, r_1)$. For each node u in RT, and each r_1 in A_u, we want to store $CC(u, r_1)$ and $CW(u, r_1)$. They can be computed in $O(\log^2 n)$ time (using $|set2(2)| = O(n)$ processors) as follows: For each u, we can compute $CC(u, r_1)$ and $CW(u, r_1)$ for all r_1 in A_u, in $O(\log n)$ time, using $|set2_u(2)|$ processors, as explained in §5.4. With $|set2(2)|$ processors, we can compute these for all the nodes u in any one level of RT, in parallel, in $O(\log n)$ time. Since RT has $O(\log n)$ height, we can compute these for all the levels in $O(\log^2 n)$ time.

Now we show how to find $T''(r2)$ and $C''(r2)$ (defined at the beginning of this subsection) using a single processor, for $r2 \in \hat{L}1(2)$. Let $dist(r, r2) = i$ and $w(v') = yvalue$, where $0 < yvalue \leq w(r2^1)$. First, process the range query $R = [1, i] \times [yvalue, \infty)$ using RT as follows. Find the allocation nodes for the x-range $[1, i]$. Let u be an allocation node. Among all the h-arcs in A_u with cutoff value less than *yvalue*, let r_u have the highest cutoff value. As explained in the previous subsection, we can find r_u for all the allocation nodes in $O(\log n)$ time. Let $cset([1, i], yvalue) = \cup\, cset(u, r_u)$, $CC([1, i], yvalue) = \sum CC(u, r_u)$, and $CW([1, i], yvalue) = \sum CW(u, r_u)$, where the union and the summations are over all the allocation nodes u. Since there are $O(\log n)$ allocation nodes, $CC([1, i], yvalue)$ and $CW([1, i], yvalue)$ can be computed in $O(\log n)$ time.

Then, using binary search, in $O(\log n)$ time, find the first h-arc $r1 \in L2(1)$ with $co(r1) < yvalue$. We let $upper(r2; yvalue)$ denote $cset(P(1), r1) \cup cset([1, i], yvalue)$. $P(r2; yvalue)$ denotes the subpolygon of $P(r2)$ that is bounded above by all $r' \in upper(r2; yvalue)$ (see Figure 10). Then, $T''(r2)$ is the triangulation of $P'(r2)$ that consists of r' and $T(r')$, $r' \in upper(r2; yvalue)$, and $F'(r2; yvalue)$. Its cost is

$$\begin{aligned} C''(r2) = &\sum_{r' \in\, upper(r2;\, yvalue)} C(r') \\ &+ w(v')\left[W(r2) - \sum_{r' \in\, upper(r2;\, yvalue)} (W(r') - w((r')^1)w((r')^2))\right] \\ = &\; CC(P(1), r1) + CC([1, i], yvalue) \\ &+ w(v')[W(r2) - CW(P(1), r1) - CW([1, i], yvalue)]. \end{aligned}$$

So, a single processor can compute $C''(r2)$ in $O(\log n)$ time.

5.9. The merge operation. For the *merge* operation, we need to obtain items (a)–(d) of §5.6 for the whole polygon P. We show how this can be done in $O(\log^2 n)$ time.

Note that for $r' \in set1(j) - \hat{L}1(2)$ ($j = 1, 2$), we have $P(r') = P(j, r')$; so $T(r') = T(j, r')$, $C(r') = C(j, r')$, and $co(r') = co(j, r')$. For $r2 \in \hat{L}1(2)$, $T(r2)$, $C(r2)$, and $co(r2)$ are obtained using procedures similar to those in §4 for unimodal polygons. The only added complication is due to the presence of the basic subtrees glued to $\hat{T}(1)$ and $\hat{T}(2)$. The basic subtrees glued to $\hat{T}(1)$ are accounted for by their contribution to $L2(1)$ and the cumulative costs and weights in $P(1)$. To account for the basic subtrees glued to $\hat{T}(2)$, we use the range tree as described in the previous subsection.

For $r2 \in \hat{L}1(2)$, let $\hat{L}1(2, r2)$ denote the list of successors of $r2$ in $\hat{L}1(2)$. To find $T(r2)$ and $C(r2)$, we need the following analogues of Lemma 4.1 and its corollary for multimodal polygons.

LEMMA 5.1. *Let $r2 \in \hat{L}1(2)$. There exists $r2' \in \hat{L}1(2, r2)$ such that the following holds* (*see Figure* 11): *Let $T_0(r2')$ be the triangulation of $P(r2')$ that consists of r' and $T(r')$, $r' \in upper\ (r2'; w((r2')^1))$, and $F(r2'; w((r2')^1))$. The triangulation of $P(r2)$ that consists of $T(2, r2; r2')$ and $T_0(r2')$ is an optimal triangulation $T(r2)$ of $P(r2)$.*

Proof. Similar to that of Lemma 4.1.

COROLLARY 5.2. *Let $r2 \in \hat{L}1(2)$. For $r2' \in \hat{L}1(2, r2)$, let $dist(r, r2') = i'$; let $r1'$ be the first element* (*from the front*) *in $L2(1)$ such that $co(r1') < w((r2')^1)$* (*see Figure* 4). *Let*

$$\begin{aligned} savings(r2') = &\; C(2, r2') + C(r) - CC(P(1), r1') - CC([1, i'], w((r2')^1)) \\ &- w((r2')^1)[W'(r2') - CW(P(1), r1') - CW([1, i'], w((r2')^1))] \end{aligned}$$

(*$savings(r)$ is taken to be* 0). *Pick $r2' \in \hat{L}1(2, r2)$ such that $savings(r2')$ is maximized. Let $T_0(r2')$ be the triangulation of $P(r2')$ that consists of r' and $T(r')$, $r' \in upper(r2'; w((r2')^1))$,*

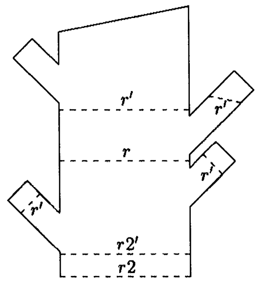


FIG. 11.

and $F(r2'; w((r2')^1))$ *(see Figure 11). The triangulation of* $P(r2)$ *that consists of* $T(2, r2; r2')$ *and* $T_0(r2')$ *is an optimal triangulation of* $P(r2)$.

Proof. The proof is similar to that of Corollary 4.2. Let $T_1(r2)$ be the triangulation of $P(r2)$ that consists of $T(2, r2)$ concatenated with $T(r)$; its cost is $C_1(r2) = C(2, r2) + C(r)$. For any $r2' \in \hat{L}1(2, r2)$, let i' and $r1'$ be as specified in the corollary. Let $T_2(r2)$ be the triangulation of $P(r2)$ specified in the corollary. It differs from $T_1(r2)$ only in $P(r2'; w((r2')^1))$. Its cost is

$$\begin{aligned} C_2(r2) &= C(2, r2; r2') + \sum_{r' \in \, upper(r2'; w((r2')^1))} C(r') \\ &\quad + w((r2')^1) \left[W'(r2') - \sum_{r' \in \, upper(r2'; w((r2')^1))} (W(r') - w((r')^1) w((r')^2)) \right] \\ &= C(2, r2; r2') + CC(P(1), r1') + CC([1, i'], w((r2')^1)) \\ &\quad + w((r2')^1)[W'(r2') - CW(P(1), r1') - CW([1, i'], w((r2')^1))] \\ &= C_1(r2) - savings(r2'). \end{aligned}$$

By Lemma 5.1, $T_2(r2)$ must be an optimal triangulation of $P(r2)$, for some $r2' \in \hat{L}1(2, r2)$. Clearly, this $r2'$ must be such that it maximizes $C_1(r2) - C_2(r2) = savings(r2')$. □

In the above corollary, if $r2' = r1' = r$, then $T(r2)$ consists of $T(2, r2)$ and $T(r)$.

Now we show how to obtain $T(r2)$ and $C(r2)$, for each $r2 \in \hat{L}1(2)$, in $O(\log n)$ time. For $r2' \in \hat{L}1(2)$, let $dist(r, r2') = i'$; let $r1' \in L2(1)$ be as specified in the above corollary. The processor assigned to $r2'$ computes $CC(P(1), r1')$, $CC([1, i'], w((r2')^1))$, $CW(P(1), r1')$, and $CW([1, i'], w((r2')^1))$ in $O(\log n)$ time as explained in the previous subsection. Then the processor computes $savings(r2')$ in constant time. Once all the processors assigned to the h-arcs in $\hat{L}1(2)$ have computed their *savings*, they perform a prefix computation from top to bottom, in $O(\log n)$ time, using the max operation, to find the largest *savings*. Among all the h-arcs in $\hat{L}1(2, r2)$, let $r2'$ have the largest *savings*; since $savings(r) = 0$, we have $savings(r2') \geq 0$. Then $T(r2)$ is the triangulation of $P(r2)$ specified in the above corollary; its cost is $C(r2) = C(2, r2) + C(r) - savings(r2')$. Thus $C(r2)$ can be obtained for all $r2 \in \hat{L}1(2)$, in $O(\log n)$ time.

Let $r2_0$ be the bottom edge (and the root) of $P(2)$ and P, and let $r2'_0$ be the h-arc in $\hat{L}1(2)$ with the largest savings. $T(r2_0)$ (as described in the previous paragraph) is an optimal triangulation of $P(r2_0) = P$, and its cost is $C(r2_0)$. $T(r2_0)$ can be obtained as follows: concatenate $T(1)$ and $T(2)$; replace the triangulation in $P(r2'_0; w(r2_0'^1))$ by a fan. So, the set *set*1 of h-arcs in $T(r2_0)$ can be obtained from *set*1(1) ∪ *set*1(2) by removing all the h-arcs of $P(r2'_0; w(r2_0'^1))$. So, *set*1 can be obtained in $O(\log n)$ time. Then the list $\hat{L}1$ of those h-arcs of $\hat{T}$ that are in *set*1 can be obtained in $O(\log n)$ time.

Now we show how to compute $co(r2)$, for each $r2 \in \hat{L}1(2)$, in $O(\log^2 n)$ time. Before we can compute $co(r2)$, we need to know which h-arcs will exist in an optimal triangulation of $P'(r2)$ when $w(v')$ is less than but arbitrarily close to $co(r2)$. $T_1'(r2)$, $C_1'(r2)$, $T_2'(r2)$, and $C_2'(r2)$ are same as defined for unimodal polygons in §4; the processor assigned to $r2$ can compute $C_1'(r2)$ in constant time. By Lemma 2.2, we have the following: for $w(v') < co(r2)$, $C_2'(r2) < C_1'(r2)$; at $w(v') = co(r2)$, $C_2'(r2) = C_1'(r2)$.

By Lemma 3.3, we have $co(r2) \geq co(2, r2)$. To determine if $co(r2) = co(2, r2)$ or $co(r2) > co(2, r2)$, we need the following analogues of Lemmas 4.3 and 4.4.

LEMMA 5.3. *For any $w(v') \leq co(r2)$, if $T_2'(r2)$ contains any h-arc below r, then $co(r2) = co(2, r2)$.*

Proof. Similar to that of Lemma 4.3.

LEMMA 5.4. *Let $r1_0$ be the first element in $L2(1)$ such that $co(r1_0) < co(2, r2)$ (see Figure 5). Let $T_0'(r2)$ be the triangulation $T''(r2)$ of $P'(r2)$ described in the previous subsection when $w(v') = co(2, r2)$; let $C_0'(r2)$ be its cost. Then $co(r2) > co(2, r2)$ iff $C_0'(r2) < C_1'(r2)$ at $w(v') = co(2, r2)$.*

Proof. Similar to that of Lemma 4.4.

The processor assigned to $r2$ can determine $r1_0$ and $C_0'(r2)$ (specified in the above lemma), in $O(\log n)$ time as described in the previous subsection. If $C_0'(r2) \geq C_1'(r2)$, the processor sets $co(r2) = co(2, r2)$. Now consider the case $C_0'(r2) < C_1'(r2)$; we have $r1_0 > r$ and $co(r2) > co(2, r2)$. $co(r2)$ lies between the cutoff values of an adjacent pair of predecessors of $r1_0$ in $L2(1)$. Also, by Lemma 5.3, when $w(v') \leq co(r2)$, $T_2'(r2)$ cannot contain any h-arc below r. The processor assigned to $r2$ can locate $co(r2)$ with respect to $L2(1)$ as follows: The processor, in a binary search manner, picks a proper predecessor $r1_1$ of $r1_0$ in $L2(1)$, with $co(r1_1) < w(r2^1)$ (see Figure 5). (If no such $r1_1$ exists, then $co(r2)$ lies between the cutoff values of $r1_0$ and its immediate predecessor in $L2(1)$). Then the processor *tentatively* assigns the value of $co(r1_1)$ to $w(v')$. Let $r1_2$ be the immediate successor of $r1_1$ in $L2(1)$. Let $T_3'(r2)$ be the triangulation $T''(r2)$ of $P'(r2)$ described in the previous subsection when $w(v') = co(r1_1)$. For $co(r1_2) < w(v') \leq co(r1_1)$, $T_3'(r2)$ is of minimum cost among all triangulations of $P'(r2)$ that do not contain any h-arc below r. The processor assigned to $r2$ computes its cost $C_3'(r2)$ at $w(v') = co(r1_1)$, in $O(\log n)$ time, as described in the previous subsection. By Lemmas 2.2 and 5.3, we have $co(r2) < w(v') = co(r1_1)$ iff $C_1'(r2) < C_3'(r2)$; also, $co(r2) = w(v') = co(r1_1)$ iff $C_1'(r2) = C_3'(r2)$. Thus, by performing binary search in $L2(1)$, in $O(\log^2 n)$ time, the processor can locate $co(r2)$ to be either equal to the cutoff value of some predecessor of $r1_0$ or properly between the cutoff values of an adjacent pair of predecessors of $r1_0$. In the former case, we are done. In the latter case, we still have to locate $co(r2)$ with respect to $set2(2)$. Let A be the secondary structure associated with the root of the range tree RT. Using a binary search procedure similar to the one above, in $O(\log^2 n)$ time, the processor can locate $co(r2)$ to be either equal to the cutoff value of some element in A or properly between the cutoff values of two adjacent elements in A. In the former case, we are done. In the latter case, we know which h-arcs will exist in an optimal triangulation of $P'(r2)$ when $w(v')$ is less than but arbitrarily close to $co(r2)$; then $co(r2)$ can be obtained by solving an equation. So, $co(r2)$ can be computed in $O(\log^2 n)$ time.

Once we have the final cutoff values, we can obtain the list $\hat{L}2$ for $\hat{T}$, in $O(\log n)$ time, as follows: By performing a parallel prefix computation in $\hat{L}1$, find the smallest cutoff value $co(r'')$ preceding each $r' \in \hat{L}1$. Then $r' \in \hat{L}2$ iff $co(r') < co(r'')$.

Once we have $\hat{L}2$, we can obtain the list $L2$ for P, in $O(\log n)$ time, as follows: For each basic subpolygon S glued to $\hat{T}$, find the topmost h-arc in $\hat{L}2$ that is below the base of S; let co_S be the cutoff value of this h-arc. Let $L2'(S)$ be the list obtained from $L2(S)$, by removing the longest prefix of elements all of which have cutoff values greater than or equal to co_S. Then $L2$ is obtained by sorting $\hat{L}2 \cup (\cup_S L2'(S))$ in $O(\log n)$ time [5].

Finally, for each $r' \in L2$, $CC(P, r')$ and $CW(P, r')$ can be computed in $O(\log n)$ time as described in §5.4. So the merge operation can be performed in $O(\log^2 n)$ time. This leads to the following.

THEOREM 5.5. *An optimal triangulation of a polygon can be found in $O(\log^4 n)$ time.*

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. L. BENTLEY, *Algorithm for Klee's rectangle problems*, unpublished notes, Department of Computer Science, Carnegie–Mellon University, Pittsburgh, PA, 1977.

[3] O. BERKMAN, D. BRESLAUER, Z. GALIL, B. SCHIEBER, AND U. VISHKIN, *Highly parallelizable problems*, in Proc. 21st Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1989, pp. 309–319.

[4] P. BRADFORD, *Efficient parallel dynamic programming*, in Proc. 30th Annual Allerton Conference on Communication Control and Computing, University of Illinois Press, Champaign, IL, 1992, pp. 185–194.

[5] R. COLE, *Parallel merge sort*, SIAM J. Comput., 17 (1988), pp. 770–785.

[6] A. CZUMAJ, *Parallel algorithm for the matrix chain product and the optimal triangulation problems*, in Proc. Symposium on Theoretical Aspects of Computer Science, Lecture Notes in Comput. Sci. 665, Springer-Verlag, New York, 1993, pp. 294–305.

[7] L. GUIBAS, H. T. KUNG, AND C. THOMPSON, *Direct VLSI implementation of combinatorial algorithms*, in Proc. Caltech Conference on VLSI, California Institute of Technology, Pasadena, CA, 1979, pp. 509–525.

[8] T. C. HU AND M. T. SHING, *Computation of matrix chain products, part* I, SIAM J. Comput, 11 (1982), pp. 362–373.

[9] ———, *Computation of matrix chain products, part* II, SIAM J. Comput, 13 (1984), pp. 228–251.

[10] J. JÁJÁ, *An Introduction to Parallel Algorithms*, Addison–Wesley, Reading, MA, 1992.

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

[12] G. S. LUEKER, *A data structure for orthogonal range queries*, in Proc. 19th Annual IEEE Symposium on Foundations of Computer Science, IEEE Press, Piscataway, NJ, 1978, pp. 28–34.

[13] F. P. PREPARATA AND M. I. SHAMOS, *Computational Geometry: An Introduction*, Springer-Verlag, New York, 1985.

[14] P. RAMANAN, *A new lower bound technique and its application: Tight lower bound for a polygon triangulation problem*, SIAM J. Comput., 23 (1994), pp. 834–851.

[15] W. RYTTER, *Note on efficient parallel computations for some dynamic programming problems*, Theoret. Comp. Sci., 59 (1988), pp. 297–307.

[16] R. E. TARJAN AND U. VISHKIN, *An efficient parallel biconnectivity algorithm*, SIAM J. Comput, 14 (1985), pp. 862–874.

[17] L. VALIANT, *Parallelism in comparison problems*, SIAM J. Comput., 4 (1975), pp. 348–355.

[18] L. VALIANT, S. SKYUM, S. BERKOWITZ, AND C. RACKOFF, *Fast parallel computation of polynomials using few processors*, SIAM J. Comput., 12 (1983), pp. 641–644.

[19] D. E. WILLARD, *Predicate-oriented database search algorithms*, Ph.D. thesis, Technical Report TR 20-78, Aiken Computational Laboratory, Harvard University, Cambridge, MA, 1978.

SIAM J. COMPUT.
Vol. 25, No. 4, pp. 894–906, August 1996

CONVERGENCE IN DISTRIBUTION FOR BEST-FIT DECREASING*

WANSOO T. RHEE[†] AND MICHEL TALAGRAND[‡]

Abstract. Consider independent random variables $X_1, \ldots, X_n$ uniformly distributed over [0, 1], and denote by B_n the number of bins needed to pack items of these sizes using the best-fit decreasing algorithm. We prove that the random variable $n^{-1/2}$ converges in distribution to a nonnormal limit. The method consists of showing that the patterns created by the algorithm exhibit some kind of convergence.

Key words. bin packing, uniform distribution, best-fit decreasing, convergence in distribution

AMS subject classifications. 60F05, 90B35

1. Introduction. The bin-packing problem requires finding the minimum number of unit-size bins needed to pack a given collection of items of sizes in [0, 1], subject to the restriction that the sum of the sizes of the items allocated to a given bin must not exceed 1. This problem has many applications and is known to be NP-complete. In the present paper, we are interested in the popular approximation algorithm best-fit decreasing (BFD). In BFD, the items to be packed are first ordered according to decreasing size. Each item is then packed in turn in the bin in which it fits the best, i.e., in which the remaining space is minimal after adding the item, opening a new bin whenever necessary. Despite the simplicity of its definition, BFD does create complicated patterns, and it exhibits anomalous behavior in the sense that decreasing the size of the items to be packed might result in using a larger number of bins. The behavior of BFD on deterministic lists can be rather complicated, if not pathological. One way to assess the relative importance of these complications is to analyze how BFD operates on random lists of items. A natural randomness assumption is that the items sizes $X_1, \ldots, X_n$ are independent and distributed according to a given probability measure μ on [0, 1]. Let us denote by $B_n = B_n(X_1, \ldots, X_n)$ the number of bins used by BFD to pack such a list of items. In [5], it is proved that, for a number $b(\mu)$ depending only on μ, the sequence of random variables (r.v.'s) B_n/n converges completely to $b(\mu)$, that is, for each $\varepsilon > 0$, $\sum_{n \geq 1} P(|B_n/n - b(\mu)| > \varepsilon) < \infty$. What is actually implicitly shown in [5] is that the patterns produced by BFD while packing the random list $X_1, \ldots, X_n$ exhibit a kind of convergence, and it can be said that [5] provides in that case a complete description of the behavior of BFD up to effects affecting $o(n)$ bins. A deeper understanding would be provided by a more precise description. For example, central-limit theorems provide a description of the situation up to effects of order $o(\sqrt{n})$. The proof of a central-limit theorem in the general framework described above appears to be a tough challenge. There are, however, special distributions for which the analysis is simple. The simplest case is when the distribution has a strictly increasing density. In that case, with probability very close to 1, all of the items of size $\leq 1/2$ fit in the bins occupied by the items of size greater than 1/2. The next simplest case is the case where the distribution has a strictly decreasing density. In that case, following the results of [1], it should be easy to show (although this has not been checked by the authors) that the wasted space is $o(\sqrt{n})$ so that the number of bins used equals $\sum_{i \leq n} X_i$ within $o(\sqrt{n})$ (or even possibly $o(1)$). The next-hardest case that we consider in the present paper is where μ is the uniform distribution. In that case, BFD used on a random list does exhibit genuinely

*Received by the editors May 6, 1994; accepted for publication (in revised form) January 4, 1995. This work was partially supported by a National Science Foundation grant.

[†]Department of Management Sciences, Ohio State University, 1775 College Road, Columbus, OH 43210-1399 (rhee.1@osu.edu).

[‡]Equipe d'Analyse, Université Paris VI, Tour 46, 4 Place Jussieu, 75230 Paris cedex 05, France and Department of Mathematics, Ohio State University, 231 West 18th Avenue, Columbus, OH 43210-1399.

complicated patterns (which it does not do in the two previously discussed examples), but these complicated patterns involve only $O(\sqrt{n})$ bins. These patterns are created by the bins such that the sum of the sizes of the first two items they receive is not around $1 - O(n^{-1/2})$. To obtain a central-limit theorem, then, the task is to prove that these patterns exhibit some kind of convergence, which is the purpose of this paper.

THEOREM 1.1. *Denote by B_n the number of bins used by BFD when packing a list of n items whose sizes are independent and uniformly distributed over* $[0, 1]$. *Then the r.v.* $n^{-1/2}(B_n - \frac{n}{2})$ *converges in distribution.*

Remarks. 1. The analysis extends to the case of the uniform distribution over $[a, 1]$ $(a > 0)$. On the other hand, what happens for the uniform distribution on $[0, a]$ (for $1/2 < a < 1$) is unclear. In particular, according to the claim of the beginning of §5 of [1], a result such as Theorem 1.1 cannot hold for the normalizing factor $n^{-1/2}$.

2. In contrast with the previous work of these authors [3], [5], it does not seem possible to give an explicit expression for the limit distribution.

The paper is organized as follows. In §2, we describe the overall approach and the main step of the proof, which is a purely deterministic result (Theorem 2.2). We then deduce Theorem 1.1 from Theorem 2.2. The approach to Theorem 2.2 is described in §3. The necessary technical details are then completed in §§4 and 5.

2. The approach. Throughout the rest of the paper, we will not distinguish between items and item sizes. In order to control the size of the items of a list $\mathbf{x} = (x_i)_{i \leq n}$, we will use the function (defined for $0 \leq t \leq 1$)

$$F_n(\mathbf{x}, t) = \frac{1}{n}\text{card}\{i \leq n; x_i \geq t\} \tag{1}$$

that counts the proportion of items $\geq t$. We note the fact that we can define $F_n(\mathbf{y}, t)$ for all lists $\mathbf{y}$ of numbers, whether they are of length n or shorter. The following theorem has proved a convenient tool in the proof of central-limit theorems.

THEOREM 2.1 (see [2]). *If $X_1, \ldots, X_n$ are independent uniformly distributed (defined on a rich enough probability space) there exists a Brownian bridge $\big(W_n(t)\big)_{0 \leq t \leq 1}$ such that the r.v.*

$$D_n = \sup\nolimits_{0 \leq t \leq 1} \left| F_n(\mathbf{X}, t) - (1 - t) - \frac{1}{\sqrt{n}} W_n(t) \right|$$

satisfies

$$P\Big(D_n \geq \frac{Ku(\log n)^2}{n}\Big) \leq \exp(-u),$$

where K is a universal constant.

This result motivates the study of the packing of a list $\mathbf{x} = (x_i)_{i \leq n}$ of items which satisfies

$$\sup\nolimits_{0 \leq t \leq 1} |F_n(\mathbf{x}, t) - (1 - t) - a\varphi(t)| \leq \delta, \tag{2}$$

where φ is a continuous function on $[0, 1]$ and a and δ are small. The essential step towards Theorem 1.1 is the following deterministic result.

THEOREM 2.2. *For each $\ell \geq 0$, there exists a number K_ℓ depending only on ℓ, and for each continuous function φ on* $[0, 1]$, *there exists a number $H_\ell(\varphi)$ depending only on ℓ and φ, such that for any list* $\mathbf{x}$ *of items that satisfies* (2), *with $\delta \geq 1/n$, the number $BFD(\mathbf{x}, \ell)$ of bins used while packing the items of size $> 2^{-\ell}$ satisfies*

$$\left| \frac{1}{n} BFD(\mathbf{x}, \ell) - \frac{1}{2} - aH_\ell(\varphi) \right| \leq \delta K_\ell. \tag{3}$$

The first reaction of the reader might be disbelief, since the bound of (3) does not depend on a. One must bear in mind, however, the fact that the very existence of a list of items that satisfies (2) for a small δ forces rather stringent conditions on $a\varphi$.

COROLLARY 2.3. *If a list* $\mathbf{x}$ *of items satisfies* (2), *the number* $BFD(\mathbf{x})$ *of bins used in packing these items satisfies*

$$\left|\frac{1}{n}BFD(\mathbf{x}) - \frac{1}{2} - aH_\ell(\varphi)\right| \le \delta(K_\ell + 4)$$
$$+ 2a\big(\sup_{t\le 2^{-\ell}}|\varphi(t)| + \sup_{t\ge 1-2^{-\ell}}|\varphi(t)|\big).$$

Before the hard work starts, it is appropriate to make a technical comment. It is very convenient to have conditions of the type of (2) only for continuous functions because, for any size $t \ne 1$, items of this given size are irrelevant, and for $0 \le t < 1$, (2) implies

$$\left|\frac{1}{n}\mathrm{card}\{i \le n; x_i > t\} - (1-t) - a\varphi(t)\right| \le \delta \tag{4}$$

so that for $u < t < 1$,

$$\left|\frac{1}{n}\mathrm{card}\{i \le n; u \le x_i \le t\} - (t-u) - a\big(\varphi(u) - \varphi(t)\big)\right| \le 2\delta. \tag{5}$$

This observation will be used repeatedly (for lists of items and lists of vacancies).

Proof of Corollary 2.3. Let us first recall some convenient terminology. In the course of the packing, the only characteristic of the bins that matters is their vacancy, i.e., the space left in the bins. Thus it is convenient to not distinguish between the list of bins and the list of their vacancies. Also, we employ the convention that the packing starts with n empty bins to ensure that we have sufficient capacity to accommodate all items (so that bins having not yet received items have vacancy one).

The basic idea of Corollary 2.3 is that packing the items of size $\ge 1-2^{-\ell}$ creates vacancies of size $\le 2^{-\ell}$ that are not affected by the packing of the items of size between $2^{-\ell}$ and $1-2^{-\ell}$. Thus the list $\mathbf{v} = (v_i)_{i\le n}$ of vacancies after the items of size $> 2^{-\ell}$ are packed satisfies (using (5))

$$\begin{aligned} t \le 2^{-\ell} \Rightarrow F_n(\mathbf{v}, t) &\ge \frac{1}{n}\mathrm{card}\{i \le n; 1 - 2^{-\ell} \le x_i \le 1 - t\} \\ &\ge (2^{-\ell} - t) - 2\delta - 2a\sup_{t\ge 1-2^{-\ell}}|\varphi(t)|. \end{aligned}$$

On the other hand, under (2), the list $\mathbf{y}$ of items $x_i \le 2^{-\ell}$ satisfies

$$t \le 2^{-\ell} \Rightarrow F_n(\mathbf{y}, t) \le (2^{-\ell} - t) + 2a\sup_{t\le 2^{-\ell}}|\varphi(t)| + 2\delta.$$

Thus to prove Corollary 2.3 it suffices to prove the following.

CLAIM. *Suppose we are given a decreasing list* $\mathbf{x} = (x_1, \dots, x_k)$ *of items and a list of vacancies* $\mathbf{v} = (v_1, \dots, v_m)$ *with* $v_1 \ge v_2 \ge \dots \ge v_m$. *Then when packing these items in a sequence of bins with vacancies* $v_1, \dots, v_m, 1, 1, \dots$, *at most* $m + D(\mathbf{x}, \mathbf{v})$ *bins will receive items, where*

$$D(\mathbf{x}, \mathbf{v}) = \sup_{t\ge 0}\big\{\mathrm{card}\{i \le k; x_i \ge t) - \mathrm{card}\{i \le m; v_i \ge t\}\big\}.$$

The proof of this is almost obvious by induction over k. If $x_1 > v_1$, we put x_1 alone in a bin with vacancy one (creating a new vacancy < 1) and the value of $D(\mathbf{x}, \mathbf{v})$ decreases by at least one. If $x_1 \leq v_1$, we do not use a new bin, and we easily see that the value of $D(\mathbf{x}, \mathbf{v})$ does not increase. □

We return to the proof of Theorem 2.2. Consider the r.v.

$$R_{n,\ell} = \sup_{t \geq 1-2^{-\ell}} |W_n(t)| + \sup_{t \leq 2^{-\ell}} |W_n(t)|.$$

Setting $\mathbf{X_n} = (X_i)_{i \leq n}$, $D'_n = \max(D_n, 1/2n)$, Corollary 2.3 implies

$$\left| BFD(\mathbf{X_n}) - \frac{n}{2} - \frac{1}{\sqrt{n}} H_\ell(W_n) \right| \leq D'_n(K_\ell + 4) + \frac{2}{\sqrt{n}} R_{n,\ell}, \tag{6}$$

where W_n denotes the random function $t \rightarrow W_n(t)$. It follows from (6) that for $\ell' > \ell$ we have

$$|H_\ell(W_n) - H_{\ell'}(W_n)| \leq \sqrt{n} D'_n (K_\ell + K_{\ell'} + 8) + 2(R_{n,\ell} + R_{n,\ell'})$$

and thus, for $u > 0$,

$$\begin{aligned} P(|H_\ell(W_n) - H_{\ell'}(W_n)| \geq u) \leq {} & P\left(\sqrt{n} D'_n (K_\ell + K_{\ell'} + 8) \geq \frac{u}{2}\right) \\ & + P\left(2(R_{n,\ell} + R_{n,\ell'}) \geq \frac{u}{2}\right). \end{aligned}$$

The important observation is now that neither the left-hand side nor the last term depend in n. Thus, using Theorem 2.1 and letting n go to infinity, we get

$$P\big(|H_\ell(W) - H_{\ell'}(W)| \geq u\big) \leq P\left(R_\ell + R_{\ell'} \geq \frac{u}{4}\right), \tag{7}$$

where $W(t)$ is a Brownian bridge, W is the random function $t \rightarrow W(t)$, and R_ℓ is defined as $R_{n,\ell}$, using $W(t)$ rather than $W_n(t)$. Now (7) implies

$$E|H_\ell(W) - H_{\ell'}(W)| \leq 4E(R_\ell + R_{\ell'}). \tag{8}$$

Since obviously $\lim_{\ell \to \infty} ER_\ell = 0$, we get that the sequence $\big(H_\ell(W)\big)_\ell$ of random variables is a Cauchy sequence in the space L_1 of integrable r.v.'s. Denoting its limit by $H(W)$, we get from (8) that

$$E|H_\ell(W) - H(W)| \leq 4ER_\ell$$

and thus, by equality of distribution,

$$E|H_\ell(W_n) - H(W_n)| \leq 4ER_\ell = 4ER_{\ell,n}.$$

Going back to (6) gives

$$E\left|\sqrt{n}\left(BFD(\mathbf{X_n}) - \frac{n}{2}\right) - H(W_n)\right| \leq (K_\ell + 4)\sqrt{n} ED'_n + 6ER_\ell.$$

Since $\lim_{n \to \infty} \sqrt{n} ED'_n = 0$, the limit superior as $n \rightarrow \infty$ of the left-hand side is $\leq 6ER_\ell$ for any ℓ. Since $\lim_{\ell \to 0} ER_\ell = 0$, the limit of the left-hand side is zero, and Theorem 1.1 follows.

3. Stages of packing. In order to prove Theorem 2.2, we must keep track of the structure of the list of vacancies as the packing progresses. We recall the convention that the packing starts with n empty bins (to ensure that there are enough bins). The basic idea was introduced in [5] that the packing can be decomposed in a number of stages, during which it approximatively has a simpler structure. The structure of the list of vacancies at the end of each stage can be expressed tractably as a function of its structure at the beginning of the stage. The task is simply to show that, under (2), this structure is determined by a and φ, with an error depending only on δ and the stage in which we are currently located. While the proof has much in common with the arguments of [5], it does not focus on exactly the same aspects and thus had to be entirely rewritten.

Throughout the rest of the paper, we fix a list $\mathbf{x}$ of items, a continuous function φ on $[0, 1]$, and numbers a and δ such that (2) holds.

PROPOSITION 3.1. *For $\ell \geq 1$, $1 \leq r \leq 2^{\ell+1}$, we can find numbers $K_{\ell,r}$ depending only on ℓ and r (but not on φ or a), numbers $\alpha_{\ell,r}$ with*

$$2^{-\ell} = \alpha_{\ell,1} \geq \alpha_{\ell,2} \cdots \geq \alpha_{\ell,2^{\ell+1}} = 2^{-\ell-1}, \tag{9}$$

and continuous functions $\psi_{\ell,r}$ on $[0, 1]$ depending only on φ, ℓ, and r, such that, after all the items $x_i > \alpha_{\ell,r}$ of the list $\mathbf{x}$ have been packed, the list $\mathbf{w} = (w_i)_{i \leq n}$ of vacancies satisfies the following:

$$t \geq 2^{-\ell-1} \Rightarrow \left| F_n(\mathbf{w}, t) - \frac{1}{2} - \left(\alpha_{\ell,r} - \min(t, \alpha_{\ell,r})\right) - a\psi_{\ell,r}(t) \right| \leq K_{\ell,r}\delta. \tag{10}$$

$$t < 2^{-\ell-1} \Rightarrow \left| F_n(\mathbf{w}, t) - \frac{1}{2} - (\alpha_{\ell,r} - t) - a\psi_{\ell,r}(t) \right| \leq K_{\ell,r}\delta 2^{-\ell-1}/t. \tag{11}$$

$$\textit{If } \alpha_{\ell,r} > 2^{-\ell-1}, \textit{ then } r\alpha_{\ell,r} \leq 1, \textit{ and if } r > 1, \tag{12}$$

$$\textit{then the function } \psi_{\ell,r} \textit{ is constant in the interval } [2^{-\ell}, r\alpha_{\ell,r}].$$

Let us start with some easy observations. First, one of the purposes of (12) is that the condition $r\alpha_{\ell,r} \leq 1$ forces $\alpha_{\ell,2^{\ell+1}} = 2^{-\ell-1}$. Second, we note that the number of bins that actually received items is $n\big(1 - F_n(\mathbf{w}, 1)\big)$. Thus Theorem 2.2 follows from Proposition 3.1, used for $\ell - 1$ rather than ℓ and for $r = 2^{\ell}$. Next, we observe that the case $\ell = 1, r = 1$ of Proposition 3.1 is obvious. Finally, we observe that if Proposition 3.1 has been proved for ℓ if and $r = 2^{\ell+1}$, it certainly holds for $\ell + 1, r = 1$, as seen by taking $K_{\ell+1,1} \geq 2K_{\ell,2^{\ell+1}}$ and $\psi_{\ell+1,1} = \psi_{\ell,2^{\ell+1}}$. Thus to prove Proposition 3.1, for a given ℓ, we must construct the numbers $\alpha_{\ell,r}$ and the functions $\psi_{\ell,r}$ by induction over r, a goal that will occupy the rest of the paper.

Thus assume that $\psi_{\ell,r}$ and $\alpha_{\ell,r}$ are constructed and $r < 2^{\ell+1}$. If $\alpha_{\ell,r} = 2^{-\ell-1}$, we set $\psi_{\ell,r+1} = \psi_{\ell,r}, \alpha_{\ell,r+1} = 2^{-\ell-1}$. (The construction of the numbers $K_{\ell,r}$ will be explained later). Thus it suffices to consider the case $\alpha_{\ell,r} > 2^{-\ell-1}$. For simplicity of notation, we set $\alpha = \alpha_{\ell,r}$ and $\psi = \psi_{\ell,r}$. We start the packing of items $x_i \leq \alpha$. This operation is called the current stage of packing. It will end as the last element $> \alpha_{\ell,r+1}$ (where $\alpha_{\ell,r+1}$ will be constructed below) is packed. As the packing progresses, items are attributed to bins, whose vacancies change. The vacancy of a bin at the beginning of the current stage will be called its *initial vacancy*.

An item $x > 2^{-\ell-1}$ attributed to a bin with vacancy $\leq 2^{-\ell}$ creates a new vacancy $< 2^{-\ell-1}$ that will play no further role in the current stage of packing. We will call an item a *surplus item* if it is attributed to a bin with an initial vacancy $> 2^{-\ell}$. We will show that, with the exception of relatively few, the surplus items are attributed r at a time to each vacancy $> 2^{-\ell}$. As the packing progresses, the size of the surplus items decreases, the initial vacancy of the bins that accept them increases, and the vacancies created in such manner increase. At the beginning of the current stage, these vacancies are smaller than the size of the items being packed, so they do not interfere with the packing process. The current stage will (essentially) end when the size of these vacancies reaches the size of the items being packed. Consider the list $\mathbf{y}$ of items $\alpha \geq y_1 \geq \cdots \geq y_m$ remaining to be packed at the beginning of the current stage and the list $2^{-\ell} \geq v_1 \geq \cdots \geq v_q$ of vacancies $w^{-\ell}$ at most $2^{-\ell}$ already created as this stage begins. Consider the *basic matching procedure* that attributes in turn, starting with y_1, each item y_i to the smallest unmatched vacancy $v_j \geq y_i$ and leaves it unmatched if no such vacancy exists. Call an item an *excess item* if it is unmatched under this procedure. It is a basic fact (see, e.g., [4]) that the number $E(t)$ of excess items of size at least t is given by

$$\begin{aligned} E(t) &= \sup\nolimits_{u \geq t}\bigl(\operatorname{card}\{i \leq m;\, y_i \geq u\} - \operatorname{card}\{i \leq q;\, v_i \geq u\}\bigr) \\ &= \max(0, \sup\nolimits_{t \leq u \leq \alpha}(\operatorname{card}\{i \leq m;\, y_i \geq a\} - \operatorname{card}\{i \leq q;\, v_i \geq u\}). \end{aligned}$$

Combining this result with (2) and (10), we see that, setting $\delta' = K_{\ell,r}\delta$ for simplicity, for $t \geq 2^{-\ell-1}$, we get

$$|n^{-1}E(t) - aA(t)| \leq 4\delta', \tag{13}$$

where

$$A(t) = \max(0, \sup\nolimits_{t \leq u \leq \alpha}(\varphi(u) - \varphi(\alpha) - (\psi(u) - \psi(2^{-\ell})))).$$

We observe that $t \to A(t)$ is continuous nonincreasing and that $A(\alpha) = 0$.

Most of the excess items will be surplus items. Most of the surplus items will fit r at a time into bins of initial vacancy $> 2^{-\ell}$. To get enough space in these bins, we need to consider all bins with initial vacancy at most $\xi(t)$, where

$$\xi(t) = \sup\left\{\xi \leq 1 : \psi(2^{-\ell}) - \psi(\xi) \leq \frac{1}{r}A(t)\right\}. \tag{14}$$

Since A is nonincreasing, ξ is nonincreasing. The continuity of ψ implies that

$$\xi(t) < 1 \Rightarrow \psi(2^{-\ell}) - \psi\bigl(\xi(t)\bigr) = \frac{1}{r}A(t). \tag{15}$$

Surplus items of size about t will fit r at a time in bins with vacancy about $\xi(t)$, creating vacancies of size about $\lambda\xi(t) - rt$. We want to stop the current stage when this size reaches t, so we define

$$\alpha_{\ell,r+1} = \inf\bigl\{t \geq 2^{-\ell-1};\, \xi(t) \leq (r+1)t\bigr\}. \tag{16}$$

For simplicity, we write $\alpha' = \alpha_{\ell,r+1}$, and we observe a few facts about α'. If $\alpha' > 2^{-\ell-1}$, for $t < \alpha'$, we have $\xi(t) > (r+1)t$ by the definition of α'. Since $\xi(t) \leq 1$, we have $\alpha'(r+1) \leq 1$ as required by (12). By the definition of $\xi(t)$, we have

$$\psi(2^{-\ell}) - \psi\bigl((r+1)t\bigr) \leq \frac{1}{r}A(t)$$

so that, by continuity of ψ and A, we have

$$\psi(2^{-\ell}) - \psi\bigl((r+1)\alpha'\bigr) \leq \frac{1}{r}A(\alpha'). \tag{17}$$

Also, for $t > \alpha'$, we have $\xi(t) \leq (r+1)t$. Thus if $(r+1)t < 1$, we have by (15) that

$$\psi(2^{-\ell}) - \psi\bigl((r+1)t\bigr) \geq \frac{1}{r}A(t).$$

Thus if $(r+1)\alpha' < 1$, we have by continuity of ψ that

$$\psi(2^{-\ell}) - \psi\big((r+1)\alpha'\big) \geq \frac{1}{r} A(\alpha'). \tag{18}$$

We now consider the packing of the items y_i of size $\alpha \geq y_i > \alpha'$. The way this packing proceeds was approximatively described above, and we now have to examine how the actual packing differs from this approximation. We call a bin *irregular* if either (a) its initial vacancy w satisfies $2^{-\ell} < w < r\alpha$ or (b) its initial vacancy w satisfies $w > (r+1)\alpha'$, and the bin receives at least one item during the current stage of packing.

We call an item *irregular* if it falls into an irregular bin, and we call it *regular* otherwise. We observe that an irregular item is a surplus item, and we proceed to establish a bound on the number S of these items. We observe that a surplus item must be an excess item, as shown by the argument of the claim that follows Corollary 2.3. Therefore, (13) implies

$$n^{-1} S \leq a A(\alpha') + 4\delta'. \tag{19}$$

Suppose that there exist irregular bins that arise because of condition (b) above. Then $(r+1)\alpha' < 1$, so that (18) holds. Combining this result with (10) and using (12), we see that the number M of bins with initial vacancy w such that $r\alpha < w < (r+1)\alpha'$ satisfies

$$n^{-1} M \geq \frac{a}{r} A(\alpha') - 2\delta'. \tag{20}$$

Since the surplus items are of size $\leq \alpha$, each vacancy $w > r\alpha$ will accept at least r consecutive such items. Therefore, combining (19) and (20), we see that at most $(2r+4)n\delta'$ surplus items will not fit into bins with initial vacancies $w < (r+1)\alpha'$, so that at most

$$r^{-1}(2r+4)n\delta' + 1 \leq 6n\delta' + 1 \leq 7n\delta'$$

irregular bins arise from condition (b). (Here we have used the fact that, since $\delta \geq 1/n$, we have $1 \leq \delta'_n$. This fact will be used several times.) Since ψ is constant on $[2^{\ell}, r\alpha]$, it follows from (10) that at most $2n\delta'$ irregular bins arise from condition (a). Thus there are at most $9n\delta'$ irregular bins and thus (rather crudely) at most $9 \cdot 2^{\ell+1} n\delta'$ irregular items.

Consider a bin whose initial vacancy w satisfies $r\alpha \leq w < (r+1)\alpha'$. After this bin has received $r-1$ items of size $\leq \alpha$, its vacancy is $\geq \alpha$. On the other hand, after this bin has received r items of size $\geq \alpha'$, its vacancy is $< \alpha'$. Let us now examine how the regular items

$$\alpha \geq z_1 \geq \cdots \geq z_q > \alpha'$$

are packed. When z_i is packed, it is attributed to the bin with the smallest vacancy w' larger than z_i. By the definition of what we call a regular item, the initial vacancy w of this bin satisfies either $w \leq 2^{-\ell}$ or $\alpha r \leq w < \alpha(r+1)$. If $w \leq 2^{-\ell}$, the bin has not yet received items in the current stage of packing. If $\alpha r \leq w < \alpha'(r+1)$, the bin has received at most $r-1$ items in the current stage of packing, for otherwise its vacancy would be $\leq w - r\alpha' \leq \alpha' < z_i$. Thus its vacancy w' before it receives z_i is at least $w - (r-1)\alpha \geq \alpha$. We now claim that all the bins with initial vacancy v with $z_i \leq v \leq 2^{-\ell}$ have already received items. This is obvious if $v \leq w'$ and, in particular, if $v \leq \alpha$. There is nothing more to show if $r = 1$, since the $\alpha = 2^{-\ell}$ and $w' \geq \alpha$. If $r \geq 2$, then $w \geq 2\alpha \geq 2^{-\ell}$, so there is nothing to show unless $w' < w$, i.e., the bin that accepts z_i has already accepted one item prior to z_i. Denote by $z_j \geq z_i$ the item it has accepted prior to z_i. Since $w - (r-2)\alpha \geq 2\alpha \geq 2^{-\ell}$, all the bins with initial vacancy v such that $z_j \leq v \leq 2^{-\ell}$ have already accepted items at the time z_j is packed. Since $z_j \leq \alpha$, this completes the argument. A consequence of this fact is that the regular items $z_1 \geq \cdots$ are packed by the following procedure: we perform the basic matching of these elements and of the list of vacancies $\leq 2^{-\ell}$. The unmatched items (that are exactly the regular surplus items) are packed r at a time in the vacancies $\geq r\alpha$.

So, at the end of the current stage of packing, the situation is as follows.

1. We have created irregular bins, for which we have no control on the final vacancy; but there are at most $9n\delta'$ of them.
2. From the list of initial vacancies $> 2^{-\ell}$, we have removed the irregular bins, as well as the bins that did receive surplus items. On the other hand, these surplus items did create new vacancies (all $< \alpha'$).
3. The basic matching of the list of regular items and of initial vacancies$< 2^{-\ell}$ has removed some vacancies and created new ones.

Thus the remaining task is to show that the new list of vacancies obtained from these operations can be controlled using ψ, to define $\psi' = \psi_{\ell,r+1}$, and to establish (10) for $r+1$. This is the purpose of §§4 and 5.

4. Surplus items. Throughout §§4 and 5, we will keep the following notations. We will denote by $\alpha \geq z_1 \geq \cdots$ the regular items that are packed in the current stage of packing, by $2^{-\ell} \geq v_1 \geq \cdots$ the initial vacancies that are at most $2^{-\ell}$, and by $\alpha r \leq u_1 \leq u_2 \cdots$ the initial vacancies that are $\geq \alpha r$.

Thus among the items z_i, those that are unmatched in the basic matching with the vacancies v_j are exactly the surplus items that are regular; since we no longer deal with irregular items, we simply call these the surplus items. We denote these by $\sigma_1 \geq \sigma_2 \geq \cdots$. Because there are at most $9 \cdot 2^{\ell+1} n\delta'$ irregular items, it follows from (13) that for $\alpha' \leq t \leq \alpha$, the number $\delta(t)$ of surplus items of size at least t satisfies

$$(21) \qquad aA(t) - 13 \cdot 2^{\ell+1}\delta' \leq aA(t) - (4 + 9 \cdot 2^{\ell+1})\delta' \leq n^{-1}\delta(t) \leq aA(t) + 4\delta'.$$

Starting with σ_1, consecutive surplus items are attributed r at a time to vacancies $u_1, u_2, \ldots,$ creating for $k \geq 1$, vacancies of size

$$(22) \qquad \beta_k = u_k - \sum_{1+(k-1)r \leq j \leq kr} \sigma_j.$$

By the definition of regular items, we do not run out of vacancies of size $\leq \alpha'(r+1)$ but possibly we run out of surplus items before all these vacancies have been used up (in that case, we will have no control on the final vacancy of the bin that receives the last surplus item). By (21), we see that there are at least $n\big(aA(\alpha') - 13 \cdot 2^{\ell+1}\delta'\big)$ surplus items. These need at least $nr^{-1}\big(aA(\alpha') - 13 \cdot 2^{\ell+1}\delta'\big)$ bins to accommodate them.

First, suppose $\alpha' > 2^{-\ell-1}$ and set $\tau = (r+1)\alpha'$. Combining (17) and (10), we see that there are at most $nr^{-1}aA(\alpha') + 2\delta' n$ bins of initial vacancy w with $2^{-\ell} < w < \tau$. Thus all but at most $15 \cdot 2^{\ell+1}\delta' n$ of these bins will be used by the surplus items.

Now suppose $\alpha' = 2^{-\ell-1}$. We define τ as

$$\tau = \sup\left\{t : \psi(2^{-\ell}) - \psi(t) \leq \frac{1}{r}A(\alpha')\right\}.$$

Thus $\psi(2^{-\ell}) - \psi(\tau) \leq \frac{1}{r}A(\alpha')$, and the above conclusion remains true.

We define the function $\psi_1(t)$ as follows.

Case 1. $\tau < 1$. We set

$$\psi_1(t) = \min\big(\psi(t), \psi(\tau)\big).$$

Case 2. $\tau = 1$. For all $0 \leq t \leq 1$, we set

$$\psi_1(t) = \psi(1) - \left(\frac{A(\alpha')}{r} - \big(\psi(2^{-\ell}) - \psi(1)\big)\right).$$

PROPOSITION 4.1. *The list* $\mathbf{w}' = (w_i')_{i \leq n}$ *of vacancies at the end of the current stage satisfies*

$$t \geq 2^{-\ell} \Rightarrow \left|F_n(\mathbf{w}', t) - \frac{1}{2} - a\psi_1(t)\right| \leq \frac{1}{n} + 14 \cdot 2^{\ell+1}\delta' \leq 15 \cdot 2^{\ell+1}\delta'.$$

We note that by definition ψ_1 is constant on the interval $[2^{-\ell}, \alpha'(r+1)]$ unless $\alpha' = 2^{-\ell-1}$.

In the case $\tau < 1$, Proposition 4.1 expresses the fact proved above that the surplus items remove from the list of vacancies $\geq 2^{-\ell}$ all the vacancies $< \tau$, with the possible exception of the last $14 \cdot 2^{\ell+1}n\delta'$ such vacancies, and remove at most $7n\delta'$ vacancies $\leq \tau$ (as is shown in the argument following (20)). The term $1/n$ is created by the last bin with initial vacancy $> 2^{-\ell}$ to receive items. In the case $\tau = 1$, the argument is similar, but the surplus items not absorbed by the bins with initial vacancy < 1 are received, r at a time, by bins of vacancy 1 (= new bins).

We now turn to the study of the vacancies created by the surplus items. Consider the function defined for $t \in [0, 1]$ by

$$\theta(t) = \inf\{x; \alpha' \leq x \leq 1, \xi(x) - rx < t\}.$$

The idea is that the surplus items of size about x create vacancies of size about $\xi(x) - rx$, and about $\eta(t) = A\big(\theta(t)\big)$ of these should be $\leq t$.

LEMMA 4.2. *The function* $\theta(t)$ *is decreasing continuous.*

Proof. It is obvious that θ is left continuous, and that θ decreases since $\xi(x) - rx$ decreases. To prove that θ is right continuous, consider t with $\theta(t) > \alpha'$, and consider $\alpha' < b < \theta(t)$. Consider $b < b' < \theta(t)$. Then $\xi(b') - rb' \geq t$, so that $\xi(b) - rb \geq \xi(b') - rb' + r(b' - b) \geq t + r(b' - b)$ and thus $\theta(t') \geq b$ if $t' < t + r(b' - b)$. □

COROLLARY 4.3. *The function* $\eta(t)$ *is nondecreasing continuous.*

Now, we find a lower bound on the number of vacancies $\beta_k < t$. Consider $x > \theta(t)$. According to (13), the number of surplus items of size $> x$ is at least

$$naA(x) - 9 \cdot 2^{\ell+1}n\delta'.$$

It follows from (10) and (15) that the number of vacancies of size $\alpha r < w \leq \xi(x)$ is at least $nr^{-1}A(x) - 2n\delta'$. Attributing r items of size $> x$ to a vacancy $\leq \xi(x)$ creates a vacancy less than $\xi(x) - rx \leq t$. Thereby the number of vacancies $\beta_k < t$ is at least $nar^{-1}A(x) - 9 \cdot 2^{\ell+1}n\delta'$. As $x > \theta(t)$ is arbitrary, this number of vacancies is at least $nar^{-1}\eta(t) - 9 \cdot 2^{\ell+1}n\delta'$.

Next, we find an upper bound on the number of vacancies $\beta_k < t$. An upper bound is always given by r^{-1} times the number of surplus items and hence by $anr^{-1}A(\alpha') + 2n\delta$. Consider

$$\xi_0 = \sup\{\xi(x) - rx; x > \alpha'\}.$$

We observe that, since the function $\xi(x) - rx$ decreases, $\xi(x) - rx < \xi_0$ for $x > \alpha'$. Thus $\theta(t) = \alpha'$ if $t \geq \xi_0$. In that case, $\eta(t) = A(\alpha')$, so the number of vacancies $< t$ created by surplus items is at most $an\eta(t) + 2n\delta'$. Consider now $t < \xi_0$. Then $\theta(t) > \alpha'$. Consider $\alpha' < x < \theta(t)$. Then the definition of $\theta(t)$ shows that $\xi(x) - rx \geq t$. Therefore, we have

$\beta_k \geq t$ unless either $u_k < \xi(x)$ or $\sigma_{1+(k-1)r} > x$. By (10) and (14), the number of values of k for which $u_k < \xi(x)$ is at most $nar^{-1}A(x) + 4n\delta'$. By (10) and (13), the number of values of k for which $\sigma_{1+(k-1)r} > x$ is at most $1 + nar^{-1}A(x) + 4n\delta'$. Thus (remembering that the sequence u_k increases and the sequence σ_j decreases) the number of values of k for which $\beta_k < t$ is at most

$$nar^{-1}A(x) + 4n\delta' + 1 \leq nar^{-1}A(x) + 5n\delta'.$$

Since $x < \theta(t)$ is arbitrary, this number is at most $nar^{-1}\eta(t) + 5n\delta'$. We have proved the following.

PROPOSITION 4.4. *For* $0 \leq t \leq 1$,

$$\left|\frac{1}{n}\operatorname{card}\{k;\ \beta_k < t\} - \frac{a}{r}\eta(t)\right| \leq 9 \cdot 2^{\ell+1}\delta'.$$

Since there are at most $9n\delta'$ irregular bins, we have the following.

COROLLARY 4.5. *The list β of vacancies of bins with initial vacancies $w > 2^{-\ell}$ that received items during the current stage of packing satisfies*

$$\forall t, 0 \leq t \leq 1, \left|F_n(\beta, t) - \frac{a}{r}\eta(1-t)\right| \leq 2^{\ell+6}\delta'.$$

5. Small vacancies. In this section, we study the vacancies created by the basic matching of the list of regular items $\alpha \geq \xi_1 \geq \cdots > \alpha'$ of size between α and α' and of the list of bins with initial vacancies $2^{-\ell} \geq v_1 > \cdots$. Some of these bins receive no items. We first study the list of these.

Let us denote by $U(t)$ the number of bins with initial vacancy $\leq t$ that receive no items. For $t \leq \alpha'$, we have

$$U(t) = \operatorname{card}\{j;\ v_j \leq t\}. \tag{23}$$

For $t \geq \alpha'$, by Lemma 1 of [5], we have

$$\begin{aligned} U(t) &= \sup\nolimits_{0 \leq u \leq t}\big(\operatorname{card}\{j;\ v_j \leq u\} - \operatorname{card}\{i;\ \xi_i \leq u\}\big) \\ &= \sup\nolimits_{\alpha' \leq u \leq t}\big(\operatorname{card}\{j;\ v_j \leq u\} - \operatorname{card}\{i;\ \xi_i \leq u\}\big). \end{aligned} \tag{24}$$

Consider the continuous function φ_1 on $[0, 1]$, given by $\varphi_1(u) = 0$ if $u \leq \alpha'$, $\varphi_1(u) = \varphi(\alpha') - \varphi(u)$ if $\alpha' \leq u \leq \alpha$, and $\varphi_1(u) = \varphi(\alpha') - \varphi(\alpha)$ is $u \geq \alpha$. Taking into account the number of irregular items, it follows from (2) that

$$\left|\frac{1}{n}\operatorname{card}\{i;\ \xi_i \leq u\} - \min\big(\alpha - \alpha', \max(u - \alpha', 0)\big) - a\varphi_1(u)\right| \leq 11 \cdot 2^{\ell+1}\delta'. \tag{25}$$

Let us now define the function $g(u)$ by $g(u) = 1$ if $u \geq 2^{-\ell-1}$ and $g(u) = 2^{-\ell-1}/u$ if $u \leq 2^{-\ell-1}$. It follows from (10) and (11), that

$$\left|\frac{1}{n}\operatorname{card}\{j;\ v_j \leq u\} - \frac{1}{2} + \alpha - \min(\alpha, u) + a\psi(u)\right| \leq 2\delta' g(u). \tag{26}$$

Combining this result with (25), we see that for $u \geq \alpha'$,

$$\left|\frac{1}{n}\big(\operatorname{card}\{j;\ v_j \leq u\} - \operatorname{card}\{i;\ \xi_i \leq u\}\big) - \frac{1}{2} + \alpha - \alpha' + a\big(\psi(u) + \varphi_1(u)\big)\right| \leq 2^{\ell+5}\delta'$$

so that by (24), for $t \geq \alpha'$, we have

$$\left|\frac{1}{n}U(t) - \frac{1}{2} + \alpha - \alpha' + a\psi_3(t)\right| \leq 2^{\ell+5}\delta',$$

where

$$\psi_3(t) = \sup\{\psi(u) + \varphi_1(u) : \alpha' \leq u \leq t\}.$$

Observe that $\psi_3(\alpha') = \phi(\alpha')$ since $\varphi_1(\alpha') = 0$.

Also, by (23) and (26), we get, for $t \leq \alpha'$,

$$\left|\frac{1}{n}U(t) - \frac{1}{2} + \alpha - t + a\psi(t)\right| \leq 2^{\ell+5}\delta' g(t).$$

Thus if we define the continuous function ψ_4 on $[0, 2^{-\ell}]$ by $\psi_4(t) = \psi(t)$ for $t \leq \alpha'$ and $\psi_4(t) = \psi_3(t)$ for $t \geq \alpha'$, we have shown that, for $0 \leq t \leq 2^{-\ell}$, we have

$$\left|\frac{1}{n}U(t) - \frac{1}{2} + \alpha - \min(\alpha', t) + a\psi_4(t)\right| \leq 2^{\ell+5}\delta' g(t).$$

Now define $\psi_5(t)$ for $0 \leq t \leq 1$ by $\psi_5(t) = 0$ for $t \geq 2^{-\ell}$ and $\psi_5(t) = \psi_4(t) - \psi_4(2^{-\ell})$ for $t \leq 2^{-\ell}$. We have proved the following.

PROPOSITION 5.1. *The lists* $\mathbf{u}$ *of vacancies of the bins with initial vacancy* $\leq 2^{-\ell}$ *that have not received items at the end of the current stage satisfies*

$$|F_n(\mathbf{u}, t) - \alpha' + \min(t, \alpha') - a\psi_5(t)| \leq 2^{\ell+6}\delta' g(t).$$

We now study vacancies that are created by attributing items to vacancies.

The basic fact [5] here is that if $x \leq v$, the number of pairs (x_i, v_j) created by the basic matching procedure that satisfy $x_i \geq x$, $v_j \leq v$ is equal to

$$N(x, v) = \inf\bigl\{\mathrm{card}\{i; x \leq \xi_i \leq t\} + \mathrm{card}\{j; t < v_j \leq v\}; x \leq t \leq v\bigr\}. \tag{27}$$

We observe that $N(x, v) = N(x, \alpha)$ if $v \geq \alpha$.

For $x \leq v \leq \alpha$, we define

$$G(x, v) = \inf\{\varphi(x) - \varphi(t) + \psi(t) - \varphi(v); x \leq t \leq v\}. \tag{28}$$

For $x \leq \alpha \leq v$, we define $G(x, v) = G(x, \alpha)$. For $v \leq x$, we define $G(x, v) = 0$. Thus $G(x, v)$ is a continuous function on the set $A = [\alpha, \alpha'] \times [\alpha, 2^{-\ell}]$. If we combine (25)–(28), we get

$$\left|\frac{1}{n}N(x, v) - \min(v, \alpha) - x - aG(x, v)\right| \leq 2^{\ell+5}\delta'. \tag{29}$$

Now consider $\alpha \leq x_1, x_2 \leq \alpha'$, $v \leq v_1, v_2 \leq 2^{-\ell}$, and the rectangle $R = [x_1, x_2] \times [v_1, v_2]$. We define

$$G(R) = G(x_1, v_2) + G(x_2, v_1) - G(x_1, v_1) - G(x_1, v_2). \tag{30}$$

If R' is a product of intervals with the same interior as R, we define $G(R') = G(R)$.

If we consider the two rectangles $R_1$and R_2 obtained from R by making a horizontal or a vertical cut, it follows from (30) that $G(R) = G(R_1) + G(R_2)$. Consequently, if a set S is a finite union of rectangles, we can define $G(S)$ in an unambiguous way—as the sum of the $G(S_i)$'s, where (S_i) is a covering of S by rectangles with nonoverlapping interiors. Moreover, it follows from (29) that if S can be written using m such rectangles, we have

$$\left|\frac{1}{n}N(S) - \mu(S) - aG(S)\right| \leq m2^{\ell+5}\delta', \tag{31}$$

where $N(S)$ denotes the number of matched couples (x_i, v_j) that belong to S and where

$$\mu(S) = |\{\alpha \leq x \leq \alpha'; (x, x) \in S\}|.$$

To prove (31), we can simply proceed by induction over m, (i) observing that when S is a product of two intervals, (31) follows from (29) and (ii) using the definition of $G(S)$.

Now consider an integer q and a sequence $\mathbf{s}$ of points $(x_1, y_1), \ldots, (x_q, y_q)$ of A. We set

$$R(\mathbf{s}) = \{(x, y) \in A; \exists m \leq q; x \leq x_m, y \geq y_m\}.$$

The map $\mathbf{s} \to G\big(R(\mathbf{s})\big)$ is continuous on A^q. This is obvious by using a natural decomposition of $R(\mathbf{s})$ into rectangles.

Given $u > 0$, consider the set $A_{q,u}$ of all sequences $\mathbf{s}$ of A^q such that $y_m \geq u + x_m$ for each $m \leq q$, and consider the function

$$h_m(u) = \sup\big\{G\big(R(\mathbf{s})\big); \mathbf{s} \in A_{m,u}\big\}.$$

It should be obvious that this is a continuous function of u.

LEMMA 5.2. *If $m \geq 1 + (\alpha - \alpha')/u$ we have*

$$N\Big(\{(x, v) \in A; v \geq x + u\}\Big) = \sup\big\{N\big(R(\mathbf{s})\big); \mathbf{s} \in R_{m,u}\big\}.$$

Proof. Step 1. Starting with $t_0 = \alpha'$, we construct the sequence t_i by induction as follows: t_{i+1} is largest such that the set

$$S_i = \big\{(x, v); t_i < x \leq t_{i+1}, t_i + u < v < t_{i+1} + u, v \geq x + u\big\}$$

contains no matched couple.

Step 2. We prove that for $t_{i+2} < \alpha$, we have $t_{i+2} \geq t_{i+1} + u$. By construction, there is a matched couple (x, v) such that $v = t_{i+1} + u$ and $t_i < x \leq t_{i+1}$ and a matched couple (x', v') such that $v' = t_{i+2} + u$ and $t_{i+1} < x' \leq t_{i+2}$. Since x' is matched to v', no vacancy of size $x' < w < v'$ remains at the time x' is matched. Since x is matched later than x' and since $v < v'$, we have $v < x'$, so that $t_{i+1} + u < t_{i+2}$.

Step 3. By Step 2, the construction stops with a last t_q such that $(q - 1)u \leq \alpha - \alpha'$ and $t_q = \alpha'$. Now consider a matched couple (x, v) with $v \geq x + u$, and consider the largest i such that $t_i < x$, so that $x \leq t_{i+1}$. Since $(x, v) \notin S_i$ by construction and since $v \geq x + u$, we have $v \geq t_{i+1} + u$. Thus $(x, v) \in R(\mathbf{s})$, where $\mathbf{s}$ is the sequence of couples $(t_i, t_i + u)$ for $0 \leq i \leq q - 1$. □

We now combine (31), Lemma 5.2, and the definition of $h_m(u)$ to see that

$$\begin{gathered} m \geq 1 + (\alpha - \alpha')/u \\ \Rightarrow |n^{-1}N\big(\{(u, v) \in A; v \geq x + u\}\big) - ah_m(u)| \leq m2^{\ell+5}\delta'. \end{gathered} \tag{32}$$

The problem now is to get a similar formula for a function $h(u)$ independent of m. So for $p \geq \ell$ and $2^{-p-1} \leq u \leq 2^{-p}$, we define $m_1 = 1 + 2^{p-\ell}$, $m_2 = 1 + 2^{p+1-\ell}$, and

$$h(u) = 2^{p+1}(2^{-p} - u)h_{m_1}(u) + 2^{p+1}(u - 2^{-p-1})h_{m_2}(u).$$

The function h is continuous, and it follows easily from (32) that

$$|n^{-1}N(\{(x, v) \in A;\, v \geq x + u\}) - ah(u)| \leq 2^7\delta' g(u).$$

We have proved the following

PROPOSITION 5.3. *The number $V'(t)$ of vacancies of size $\geq t$ created by the bins of the list (v_i) that receive items satisfies*

$$\left|\frac{1}{n}V'(t) - ah(u)\right| \leq 2^7\delta' g(u).$$

If we combine Proposition 4.1, Corollary 4.5, and then Propositions 5.1 and 5.3, we see that the proof is complete, provided one sets $K_{\ell,r+1} = 2^9 K_{\ell,r}$. □

REFERENCES

[1] J. L. BENTLEY, D. S. JOHNSON, F. T. LEIGHTON, C. C. MCGEOCH, AND L. A. MCGEOCH, *Some unexpected results for bin packing*, Sixteenth Annual Symposium on Theory of Computing, Association for Computing Machinery, New York, 1994, pp. 279–288.

[2] J. KOMLÓS, P. MAJOR, AND G. TUSNÁDY, *An approximation of partial sums of independent random variables and the sample DFI,* Z. Wahrscheinlichkeitstheorie und Verw. Gebiete, 32 (1975), pp. 111–131.

[3] W. RHEE, *Stochastic analysis of a modified first fit decreasing,* Math. Oper. Res., 16 (1991), pp. 162–175.

[4] W. RHEE AND M. TALAGRAND, *Optimal bin packing with items of random size* III, SIAM J. Comput., 18 (1989), pp. 473–483.

[5] ———, *The complete convergence of best fit decreasing*, SIAM J. Comput., 18 (1989), pp. 909–918.

SIAM J. COMPUT.
Vol. 25, No. 5, pp. 907–935, October 1996

001

ALPHABET-INDEPENDENT TWO-DIMENSIONAL WITNESS COMPUTATION*

ZVI GALIL† AND KUNSOO PARK‡

Abstract. We study two-dimensional periodicity, introduced by Amir and Benson. We characterize periods of a two-dimensional array, namely, the vectors such that two copies of the array, one shifted by the vector over the other, overlap without a mismatch.

Using this characterization, we design an alphabet-independent linear-time algorithm for two-dimensional witness computation, i.e., an $O(m^2)$-time algorithm that finds periods of an $m \times m$ array as well as witnesses against nonperiods of the array among the vectors whose length is less than $m/4$. The constant in the O notation does not depend on the alphabet size. Combined with the alphabet-independent text-processing algorithm of Amir, Benson, and Farach [*SIAM J. Comput.*, 23 (1994), pp. 313–323], this leads to the first alphabet-independent linear-time algorithm for two-dimensional pattern matching.

Key words. two-dimensional periodicity, pattern matching, witness computation

AMS subject classifications. 68Q20, 68Q25, 68U10

1. Introduction.

The two-dimensional pattern-matching problem is as follows: Given pattern $P[0..m-1, 0..m-1]$ and text $T[0..n-1, 0..n-1]$, find all occurrences of P in T. The pattern and the text contain symbols from an alphabet Σ. Let $\sigma = \min(|\Sigma|, m^2)$. This problem and all algorithms mentioned below (including ours) can be easily generalized to rectangular arrays. Applications of the problem include computer vision [2, 4] and multimedia systems where two-dimensional images are stored in a database.

Karp, Miller, and Rosenberg [9] gave an $O((m^2+n^2)\log m)$-time algorithm for two-dimensional pattern matching as an extension of their algorithm for string matching. Then Baker [6] and Bird [7] independently gave an $O((m^2+n^2)\log \sigma)$-time algorithm (which we call the BB algorithm for short) for two-dimensional pattern matching which uses the Knuth–Morris–Pratt (KMP) algorithm [11] and the Aho–Corasick algorithm [1] for one-dimensional string matching. Since the Aho–Corasick algorithm requires a totally ordered alphabet and its time depends on the alphabet size, the same holds for the BB algorithm. For an unbounded alphabet, the BB algorithm takes $O((m^2+n^2)\log m)$ time. Using suffix trees, Amir, Landau, and Vishkin [5] also gave an algorithm whose time complexity is the same as that of the BB algorithm. Zhu and Takaoka [14] developed a randomized algorithm that uses the KMP algorithm and the Karp–Rabin algorithm [10]. Recently, Amir, Benson, and Farach [4] gave an algorithm (which we call the ABF algorithm for short) whose text processing is independent of the alphabet and takes $O(n^2)$ time, but whose pattern processing is still dependent on the alphabet and takes $O(m^2 \log \sigma)$ time. The pattern processing required by the ABF algorithm is the two-dimensional witness computation, i.e., the

*Received by the editors December 28, 1992; accepted for publication (in revised form) January 9, 1995.

†Department of Computer Science, Columbia University, 450 Computer Science Building, New York, NY 10027 (galil@cs.columbia.edu). The work of this author was supported by NSF grant CCR-9014605 and NSF CISE Institutional Infrastructure grant CDA-90-24735.

‡Department of Computer Engineering, Seoul National University, Seoul 151-742, Korea (kpark @theory.snu.ac.kr). The research of this author was supported by KOSEF grant 951-0906-069-2.

problem of finding periods of a two-dimensional array as well as witnesses against nonperiods of the array.

All previous algorithms except the ABF algorithm reduce the two-dimensional problem into one-dimensional string matching and use known techniques in string matching. The ABF algorithm uses two-dimensional periodicity for text processing, but their witness computation resorts to one-dimensional techniques. We present the first alphabet-independent linear $O(m^2)$-time algorithm for the witness computation using two-dimensional techniques. As in the KMP algorithm, the only operation on the alphabet is the equality test of two symbols. This leads to the first alphabet-independent linear-time algorithm for two-dimensional pattern matching.

In one-dimensional strings, periodicity is easy to define: a periodic string is produced by many copies of a period which are concatenated together. This concept can be generalized to two dimensions, where a rectangular array is produced by many copies of a parallelogram. However, since we deal with limited-size rectangular arrays, there can be other kinds of periodicities in two dimensions. Amir and Benson [2] studied two-dimensional periodicity and classified two-dimensional arrays into four categories: nonperiodic, lattice-periodic, line-periodic, and radiant-periodic. They also gave an $O(m^2 \log \sigma)$-time algorithm for classifying an array, which was used in the pattern processing of the ABF algorithm. We adopt the four categories and characterize the positions of an array where another copy of the array overlaps without a mismatch in each category (especially in the cases of line-periodicity and radiant-periodicity, which was not done in [2]). Our witness computation classifies an array into one of the four categories in $O(m^2)$ time, which also improves the pattern processing of two-dimensional run-length compressed matching [3].

In §2, some definitions are presented. In §3, we continue the study of two-dimensional periodicity started by Amir and Benson. We obtain new properties of two-dimensional periodicity. Theorem 1 below is a strengthening of a theorem in [2, 3] and Theorems 2 and 3 are new. The results of this section may be of independent interest. In §4, we describe the witness computation for an $m \times m$ array P. In particular, we find for each vector v with $|v| < m/4$ whether it is a period of P, and in case it is not a period, we find a witness against it. In the witness computation, we use computed witnesses to compute new witnesses. Although the algorithmic part is quite simple, the main technical difficulty is in proving that whenever we use witnesses, the elements with which we make comparisons lie inside a certain place in the array.

2. Preliminaries. Most of our terminology is from [2]. Let A be an $m \times m$ square array with rows $0, \ldots, m-1$ and columns $0, \ldots, m-1$ (i.e., $A[0..m-1, 0..m-1]$), where the upper-left corner is $A[0,0]$. A *point* is a pair of integers (i,j) for a row number i and a column number j. Note the difference in the definition of points from the x, y coordinates of the plane. We use (i,j) to represent either a point or a vector whose tail is at the origin $(0,0)$ and whose head is at (i,j). We say that a point (i,j) is *in* A (or $(i,j) \in A$) if $0 \leq i < m$ and $0 \leq j < m$. An *element* of A is $A[i,j]$ for some $(i,j) \in A$. Each element of A is a symbol from an alphabet Σ. When $u = (i,j)$, $A[i,j]$ is also denoted by $A[u]$. The *forward diagonal* (*backward diagonal*) d of A is the set of points $(i,j) \in A$ such that $j - i = d$ $(i + j = d)$.

We divide A into four quadrants of size $\lceil m/2 \rceil \times \lceil m/2 \rceil$, labeled counterclockwise from upper-left: quadrants I, II, III, and IV. Note that if m is odd, the middle row and column are overlapped among quadrants. A vector (r,c) is a *quad*-I *vector* (*quad*-III *vector*) if $r \geq 0$ and $c > 0$ ($r \leq 0$ and $c < 0$). Similarly, (r,c) is a *quad*-II *vector* (*quad*-IV *vector*) if $r < 0$ and $c \geq 0$ ($r > 0$ and $c \leq 0$). Note that quad-I (quad-II) vectors

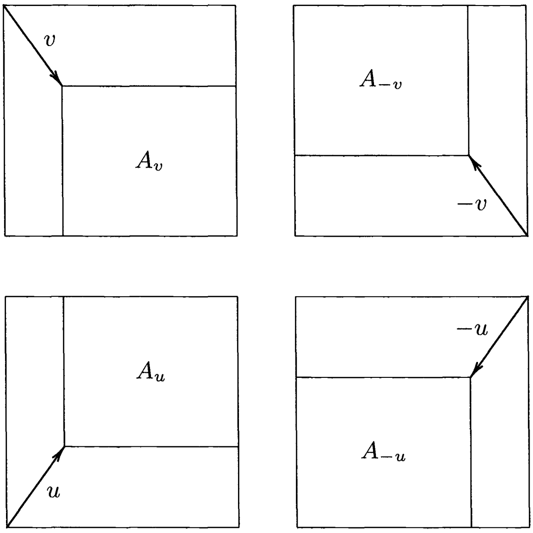


FIG. 1. *A_v and A_{-v} for a quad-*I *vector v (top). A_u and A_{-u} for a quad-*II *vector u (bottom).*

include horizontal (vertical) vectors with $r = 0$ ($c = 0$). The *length* $|v|$ of a vector $v = (r, c)$ is the maximum of the absolute values of its coordinates (i.e., $\max(|r|, |c|)$). Since there can be many vectors of the same length, we extend the notion of length to the lexicographic order of the triple $[|v|, r, c]$ ($[|v|, c, r]$) for quad-I (quad-II) vectors $v = (r, c)$. For any vector v, let A_v be the subrectangle of A consisting of all points $u \in A$ such that $u - v \in A$. See Fig. 1. We define the following partial orders on points. Let $u = (i, j)$ and $v = (k, l)$.

1. $u \prec_{\mathrm{I}} v$ (or $v \succ_{\mathrm{I}} u$) if $i \le k$ and $j < l$ (i.e., $v - u$ is a quad-I vector).
2. $u \prec_{\mathrm{II}} v$ (or $v \succ_{\mathrm{II}} u$) if $i > k$ and $j \le l$ (i.e., $v - u$ is a quad-II vector).

A sequence of points $u_1, \ldots, u_p$ in A is called a *monotone line* if $u_i \prec_{\mathrm{I}} u_{i+1}$ ($u_i \prec_{\mathrm{II}} u_{i+1}$ in quad-II) for $1 \le i \le p - 1$. For two quad-I vectors v_1 and v_2, we say that v_1 is *counterclockwise* with respect to v_2 if $r_1 c_2 < r_2 c_1$ (v_1 becomes parallel to v_2 when v_1 is rotated clockwise by less than 90°).

Let v_1 and v_2 be quad-I and quad-II vectors, respectively. The *unit cell on v_1, v_2* is the set of points $u = (r, c)$ such that $u = \alpha v_1 + \beta v_2$ for some $0 \le \alpha < 1$ and $0 \le \beta < 1$ (i.e., r and c are integers for which such α and β exist). The *lattice cell at v on v_1, v_2* is the set of points $v + u$ for u in the unit cell (i.e., the unit cell shifted by v). Two points u and v are *lattice-congruent modulo v_1, v_2* if $u - v = iv_1 + jv_2$ for integers i, j. A point lattice-congruent to $(0, 0)$ modulo v_1, v_2 is called a *lattice point on v_1, v_2*.

FACT 1. *Let v_1 and v_2 be quad-*I *and quad-*II *vectors, respectively. Every point in A is lattice-congruent to a point in the unit cell on v_1, v_2.*

We now define periodicities of square array A. A *period* of A is a vector such that two copies of A, one shifted by the vector over the other, overlap without a mismatch. Formally, a vector v is a period of A if $A_{-v} = A_v$ (i.e., for all $w \in A_{-v}$, $A[w] = A[w + v]$). A period v is *valid* if $|v| < m/4$ (i.e., $|v| \le \lceil m/4 \rceil - 1$). Let C_{I} (C_{II}) be the set of all quad-I (quad-II) vectors v such that $|v| < m/4$. We will work with quad-I and quad-II periods only, because quad-I (quad-II) periodicity implies quad-III (quad-IV) periodicity and vice versa. See Fig. 1.

We classify square arrays into the following four categories by the existence of

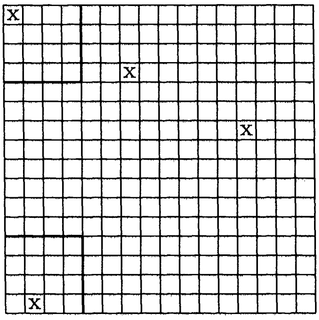


FIG. 2. *Nonperiodic.*

valid periods.

1. Nonperiodic: there are no valid periods.
2. Lattice-periodic: there exist a valid quad-I period and a valid quad-II period.
3. Line-periodic: one quadrant has no valid periods, and the other has valid periods that are on a line going through $(0, 0)$.
4. Radiant-periodic: one quadrant has no valid periods, and the other has at least two independent (vector independence) valid periods.

This classification is very similar but not identical in the line-periodic and radiant-periodic cases to the one introduced in [2, 3], where an array is line-periodic if all valid periods are of the form *iv* for the shortest valid period *v* and *integer i*. (Thus the array in Fig. 4 is not line-periodic but radiant-periodic in [2, 3].)

Although we consider square arrays for simplicity, two-dimensional periodicity in §3 and our witness computation in §4 can be extended to rectangular arrays.

3. Two-dimensional periodicity. In this section, we will characterize all the valid quad-I and quad-II periods in each of the four categories. Let $v_{\rm I}$ and $v_{\rm II}$ be the shortest quad-I and quad-II periods of A, respectively. If A is nonperiodic (i.e., $|v_{\rm I}|, |v_{\rm II}| \geq m/4$), there are no valid quad-I and quad-II periods. See Fig. 2. In all the examples of arrays, the alphabet $\Sigma = \{{\rm x}, {\rm o}, b\}$, where b is the blank symbol. The marked upper-left $\lceil m/4 \rceil \times \lceil m/4 \rceil$ square (except column 0) in Fig. 2 consists of all the points on which $(0, 0)$ is placed by a vector in $C_{\rm I}$ (and thus the points in this square comprise $C_{\rm I}$). The marked lower-left $\lceil m/4 \rceil \times \lceil m/4 \rceil$ square (except row $m-1$) consists of all the points on which $(m-1, 0)$ is placed by a vector in $C_{\rm II}$ (and thus the points in this square are not elements of $C_{\rm II}$).

LEMMA 1. *If v_1 is a quad-*I *(quad-*II*) period of A and v_2 is a quad-*I *(quad-*II*) period of A_{v_1}, then $v_1 + v_2$ is a quad-*I *(quad-*II*) period of A.*

Proof. Let $v = v_1 + v_2$ and w be a point in A_{-v}. Then $w + v_1$ and $w + v_1 + v_2$ are in A_{v_1}. Since $A[w] = A[w + v_1]$ by period v_1 of A and $A[w + v_1] = A[w + v_1 + v_2]$ by period v_2 of A_{v_1}, we have $A[w] = A[w + v_1 + v_2]$. Thus v is a period of A. □

LEMMA 2. *If v_1 and v_2 are quad-*I *(quad-*II*) periods of A such that $v_2 - v_1$ is a quad-*I *(quad-*II*) vector, then $v_2 - v_1$ is a quad-*I *(quad-*II*) period of A_{v_1}.*

Proof. Let $v = v_2 - v_1$, $B = A_{v_1}$, and w be a point in B_{-v}. Then $w - v_1$ is in A and $w - v_1 + v_2 (= w + v)$ is in B. Since $A[w] = A[w - v_1] = A[w - v_1 + v_2]$ by periods v_1, v_2 of A, $v (= v_2 - v_1)$ is a period of $B = A_{v_1}$. □

LEMMA 3. *If v_1 and v_2 are quad-*I *(quad-*II*) periods of A such that $v_2 - v_1$ is a quad-*II *(quad-*I*) vector, then $v_2 - v_1$ is a quad-*II *(quad-*I*) period of A_{v_1} and also of A_{v_2}.*

Proof. Let $v = v_2 - v_1$, $B = A_{v_1}$, and w be a point in B_{-v}. Then $w - v_1$ is in A (since $w \in B$) and $w - v_1 + v_2 (= w + v)$ is in B. Since $A[w] = A[w - v_1] = A[w - v_1 + v_2]$, $v (= v_2 - v_1)$ is a period of $B = A_{v_1}$.

Let $B' = A_{v_2}$ and w' be a point in B'_{-v}. Then $w' - v_1 + v_2 = w' + v$ is in $B' = A_{v_2}$ and thus $w' - v_1 = (w' - v_1 + v_2) - v_2$ is in A. Since $A[w'] = A[w' - v_1] = A[w' - v_1 + v_2]$, $v (= v_2 - v_1)$ is a period of $B' = A_{v_2}$. □

LEMMA 4 (see [2]). *If $v_1 = (r_1, c_1)$ and $v_2 = (r_2, c_2)$ are quad-*I *and quad-*II *periods of A, respectively, such that $r_1 + |r_2| < m$ and $c_1 + c_2 < m$, then $v_1 + v_2$ is either a quad-*I *period of A (if $r_1 \geq |r_2|$) or a quad-*II *period of A (if $r_1 < |r_2|$).*

Proof. Let $v = v_1 + v_2$ and w be a point in A_{-v}. Then $w + v_1 + v_2$ is in A, and one of $w + v_1$ and $w + v_2$ is in A because $r_1 + |r_2| < m$. Without loss of generality, assume that $w + v_1$ is in A. Since $A[w] = A[w + v_1] = A[w + v_1 + v_2]$, v is a period of A. □

LEMMA 5 (see [2]). *If $v_1 = (r_1, c_1)$ and $v_2 = (r_2, c_2)$ are quad-*I *and quad-*II *periods of A, respectively, such that $r_1 + |r_2| < m$ and $c_1 + c_2 < m$, then all lattice points $v = iv_1 + jv_2$ for $|v| < m$ are periods of A.*

Proof. By Lemma 1, all iv_1 and all jv_2 are periods of A. We partition the other lattice points into four groups divided by lines iv_1 and jv_2. We prove for the group of lattice points $iv_1 + jv_2$ such that $i, j > 0$ by induction on $d = i + j$ (the other cases are similar). When $d = 2$, $v_1 + v_2$ is a period by Lemma 4. Assume that lattice points $i'v_1 + j'v_2$ for $i' + j' < d$ are periods of A.

We now prove for $d > 2$. Consider a lattice point $v = iv_1 + jv_2$ such that $i + j = d$. By assumption, $v' = (i-1)v_1 + jv_2$ and $v'' = iv_1 + (j-1)v_2$ are periods. Since $i, j > 0$, v is either quad-I or quad-II vector. We consider the case in which v is a quad-I vector. The other case is analogous. Let $v = (r, c)$, $v' = (r', c')$, $v'' = (r'', c'')$, and $\hat{v} = (i-1)v_1 + (j-1)v_2$. Note that v' and $\hat{v}$ can be either quad-I or quad-II vectors but that v'' is a quad-I vector.

Case 1: *v' is a quad-*I *vector.* v is a period by applying Lemma 1 to v' and v_1.

Case 2: *v' is a quad-*II *vector.*

Case 2.1: *$\hat{v}$ is a quad-*I *vector.* Since $v' = \hat{v} + v_2$ is a quad-II vector, we have $|r'| < |r_2|$. Thus $r_1 + |r'| < r_1 + |r_2| < m$, and by applying Lemma 4 to v' and v_1, v is a period.

Case 2.2: *$\hat{v}$ is a quad-*II *vector.* Since $v'' = \hat{v} + v_1$ is a quad-I vector, we have $r'' < r_1$. Thus $r'' + |r_2| < r_1 + |r_2| < m$, and by applying Lemma 4 to v'' and v_2, v is a period. □

COROLLARY 1. *Let $v_1 = (r_1, c_1)$ and $v_2 = (r_2, c_2)$ be quad-*I *and quad-*II *periods of A, respectively, such that $r_1 + |r_2| < m$ and $c_1 + c_2 < m$. Then two points $u_1, u_2 \in A$ which are lattice-congruent modulo v_1, v_2 satisfy $A[u_1] = A[u_2]$.*

Proof. It follows from the fact that all lattice points are periods. □

DEFINITION. *If an array A has v_1 and v_2 that satisfy the conditions of Lemma* 5, *the whole array A can be generated by the unit cell. Such an array A is called* lattice-generative. *The shortest quad-*I *and quad-*II *periods of a lattice-generative array A are called the* basis vectors *of A.*

3.1. Lattice-periodicity.

DEFINITION. *Let $v_1 = (r_1, c_1)$ and $v_2 = (r_2, c_2)$ be quad-*I *and quad-*II *vectors, respectively. If a rectangle R contains a* $\max(r_1, |r_2|) \times (c_1 + c_2)$ *rectangle or* $(r_1 + |r_2|) \times \max(c_1, c_2)$ *rectangle, we say that R* covers *the unit cell on v_1, v_2.*

LEMMA 6. *Let R be a rectangle in the plane which covers the unit cell on v_1, v_2. Then for each point v in the unit cell on v_1, v_2, there exists a point $u \in R$ which is*

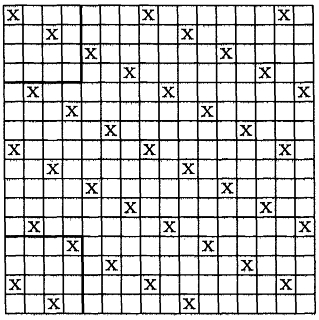


FIG. 3. *Lattice-periodic:* $v_{\rm I} = (1,2)$, $v_{\rm II} = (-3,1)$.

lattice-congruent to v *modulo* v_1, v_2.

Proof. We prove the lemma for a rectangle whose size is $\max(r_1, |r_2|) \times (c_1 + c_2)$ or $(r_1 + |r_2|) \times \max(c_1, c_2)$. Consider a $\max(r_1, |r_2|) \times (c_1 + c_2)$ rectangle R' (the other case is similar). Let $x = \max(r_1, |r_2|)$, $y = c_1 + c_2$, and E be the plane. We first show that the lemma holds for $R'_0 = E[0..x-1, 0..y-1]$. Consider each point $v = (r,c)$ in the unit cell.

Case 1: $r = 0$. Since $c < c_1 + c_2$, v is in R'_0.

Case 2: $r > 0$. Since $r < r_1$ and $c < c_1 + c_2$, v is in R'_0.

Case 3: $r < 0$. Let $i \geq 0$ be the integer such that $c_2 \leq c + ic_1 < c_1 + c_2$. If $0 \leq r + ir_1 < r_1$, then $v + iv_1$ is in R'_0 and we are done. If $r + ir_1 \geq r_1$, there is a point $v + jv_1$ for $0 < j < i$ such that $0 \leq r + jr_1 < r_1$ because $r < 0$. Since $0 < c + jc_1 < c + ic_1 < c_1 + c_2$, $v + jv_1$ is in R'_0 and we are done. Thus the remaining case is when $r + ir_1 < 0$. We show that $v + iv_1 - v_2$ is in R'_0. Since v is in the unit cell, $r_2 < r (\leq r + ir_1)$. Hence we have $r_2 < r + ir_1 < 0$. Since $0 < r + ir_1 - r_2 < |r_2|$ and $0 \leq c + ic_1 - c_2 < c_1$, $v + iv_1 - v_2$ is in R'_0.

Let $R' = E[i..i + x - 1, j..j + y - 1]$ and $L_{R'}$ be the lattice cell at (i,j). Each point v in the unit cell is obviously lattice-congruent to a point $u \in L_{R'}$, and each point $u \in L_{R'}$ is lattice-congruent to a point $w \in R'$ as in R'_0. □

COROLLARY 2. *Let* v_1 *and* v_2 *be quad*-I *and quad*-II *vectors, respectively, such that* $|v_1|, |v_2| < m/4$. *Then every point in* A *is lattice-congruent modulo* v_1 *and* v_2 *to a point in* $C_{\rm I} \cup C_{\rm II}$.

Proof. It follows from Fact 1 and the fact that $C_{\rm I} \cup C_{\rm II}$ covers the unit cell on $v_{\rm I}, v_{\rm II}$. □

THEOREM 1. *Let* v_1 *and* v_2 *be the basis vectors of a lattice-generative array* A. *Let* $z = (x,y)$ *be a quad*-I (*quad*-II) *vector of* A *such that* A_z *covers the unit cell on* v_1, v_2. *Then a quad*-I (*quad*-II) *vector* $v = (r,c)$ *for* $r \leq x$ *and* $c \leq y$ ($r \geq x$ *and* $c \leq y$) *is a period of* A *if and only if* v *is a lattice point on* v_1, v_2.

Proof: (if). This is immediate from Lemma 5. See Fig. 3.

(only if). We prove the theorem for quad-I (the proof for quad-II is similar). Suppose that $v = (r,c)$ for $r \leq x$ and $c \leq y$ is a quad-I period of A and is not a lattice point on v_1, v_2. Let u be the lattice point such that the lattice cell at u contains v. Let $u_1 = u + v_1$, $u_2 = u + v_2$, and $u_3 = u + v_1 + v_2$. Then $v - u, v - u_1, v - u_2, v - u_3$ are periods of A. We will show only that the first is a period. The other cases are similar.

We show that for every $w \in A_{-v+u}$, $A[w] = A[w + v - u]$. Since A_v contains A_z, A_v (also A_{-v}) covers the unit cell on v_1, v_2. By Fact 1 and Lemma 6, there

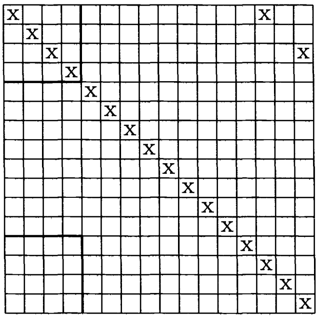


FIG. 4. *Line-periodic: valid quad-*I *periods* $(2,2),(3,3)$.

is w' in A_{-v} which is lattice-congruent to w, and $A[w'] = A[w]$ by Corollary 1. $A[w'] = A[w'+v]$ because v is a period of A. Since u is a lattice point, $w'+v$ is lattice-congruent to $w+v-u$. By Corollary 1, $A[w'+v] = A[w+v-u]$. Therefore, we have $A[w] = A[w'] = A[w'+v] = A[w+v-u]$.

Connecting v to the four corners of the lattice cell at u, we get four triangles. Choose one with an obtuse or right angle at v. Then the parallelogram side of this triangle (say it is v_1) is longer than the other two sides (one of $v-u_1$ and $v-u_2$ and one of $v-u$ and $v-u_3$). Since at least one of the two is a quad-I (or quad-III) vector, we have a shorter quad-I period than v_1. This is a contradiction. □

COROLLARY 3. *If A is lattice-periodic with basis vectors v_{I} and v_{II}, a vector $v \in C_{\mathrm{I}}$ $(v \in C_{\mathrm{II}})$ is a valid quad-*I *period* (*quad-*II *period*) *if and only if v is a lattice point on $v_{\mathrm{I}}, v_{\mathrm{II}}$.*

A theorem somewhat weaker than Theorem 1 was proved in [2], from which Corollary 3 follows. We will later apply Theorem 1 several times. In some of these applications, the version of [2] is not sufficient. We will use Theorem 1 as follows. When we say "by Theorem 1 with A, basis vectors v_1, v_2, and $z = (x,y)$," we will mean the following: Let $z = (x,y)$. Then A_z covers the unit cell on v_1, v_2. Therefore, Theorem 1 applies.

3.2. Line-periodicity. If A is line-periodic or radiant-periodic, one of v_{I} and v_{II} is a period and the other is not. If v_{I} (v_{II}) is a period, we say that A is line-periodic or radiant-periodic *in quad-*I (*in quad-*II). When A is line-periodic in quad-I, all valid periods are on the line passing through $(0,0)$ and v_{I}. We show that the distances between valid quad-I periods are nonincreasing.

THEOREM 2. *Let $s_1, \ldots, s_p$ be the valid quad-*I *periods of A from shortest to longest, and let $s_0 = (0,0)$. Then $|s_i - s_{i-1}| \geq |s_{i+1} - s_i|$ for $1 \leq i \leq p-1$.*

Proof. Since $s_i - s_{i-1}$ is a quad-I period of $A_{s_{i-1}}$ by Lemma 2, so is $2(s_i - s_{i-1})$. By Lemma 1, $2s_i - s_{i-1} (= s_{i-1} + 2(s_i - s_{i-1}))$ is a quad-I period of A. Since s_{i+1} is shorter than or equal to $2s_i - s_{i-1}$, we have $|s_i - s_{i-1}| \geq |s_{i+1} - s_i|$. The example in Fig. 4 shows that the inequality can be strict. □

3.3. Radiant-periodicity. When A is radiant-periodic in quad-I, there is at least one valid quad-I period $\hat{v}_{\mathrm{I}}$ that is independent of v_{I}.

LEMMA 7. *If A is radiant-periodic in quad-*I, *there exists a point $\hat{w} \in A$ such that*

(1) $A_{-\hat{w}} = A_{\hat{w}}$,

(2) *$A_{\hat{w}}$ is lattice-generative,*

(3) *the basis vectors $b_1 = (x_1, y_1)$ and $b_2 = (x_2, y_2)$ of $A_{\hat{w}}$ satisfy $|b_1|, |b_2| < m/4$,*

(4) $A_{\hat{w}}$ *contains an* $\lceil m/2\rceil \times (y_1 + y_2 + \lceil m/2\rceil)$ *rectangle.*

Proof. We describe a recursive algorithm FINDLATTICE(R, u, v) for two independent quad-I periods u, v of a rectangle R such that u is shorter than v. We call it with parameters $A, v_{\mathrm{I}}, \hat{v}_{\mathrm{I}}$, and it will find a point $\hat{w} \in A$ such that $A_{\hat{w}}$ is lattice-generative. The min and max operations below take shorter and longer vectors from the two, respectively.

```
PROCEDURE FINDLATTICE($R, u, v$)
  if $v - u$ or $u - v$ is a quad-II vector then stop;
  else ($v - u$ is a quad-I vector)
    $u' = \min\{u, v - u\}$;
    $v' = \max\{u, v - u\}$;
    FINDLATTICE($R_u, u', v'$);
  end if
end
```

If u, v are quad-I periods of R such that $v - u$ is a quad-I vector, then u and $v - u$ are quad-I periods of R_u by Lemma 2. Also, since u and v are two independent vectors, so are u and $v - u$.

Define *size*(R) to be the pair (p, q) of lengths of the sides of R. For $u = (r_1, c_1)$ and $v = (r_2, c_2)$, define *size*(u, v) to be the pair $(r_1 + r_2, c_1 + c_2)$. Notice that in FINDLATTICE, the decrease of *size*(R) is the same as the decrease of *size*(u, v). Since *size*$(A) = (m, m)$ and *size*$(v_{\mathrm{I}}, \hat{v}_{\mathrm{I}}) < (m/2, m/2)$ (inequality in each component), FINDLATTICE must stop. Let $\hat{R}, \hat{u}, \hat{v}$ be the parameters at the last call of FINDLATTICE (i.e., $\hat{v} - \hat{u}$ or $\hat{u} - \hat{v}$ is a quad-II vector). Let *size*$(\hat{R}) = (p, q)$, $\hat{u} = (r_1, c_1)$, and $\hat{v} = (r_2, c_2)$. Since $(m, m) - \mathit{size}(\hat{R}) = \mathit{size}(v_{\mathrm{I}}, \hat{v}_{\mathrm{I}}) - \mathit{size}(\hat{u}, \hat{v}) < (m/2, m/2) - \mathit{size}(\hat{u}, \hat{v})$, we have $\mathit{size}(\hat{R}) > \mathit{size}(\hat{u}, \hat{v}) + (m/2, m/2)$, and therefore $p \geq r_1 + r_2 + \lceil m/2\rceil$ and $q \geq c_1 + c_2 + \lceil m/2\rceil$. Assume that $\hat{v} - \hat{u}$ is a quad-II vector (the other case is similar). We consider $\hat{R}_{\hat{u}}$. Let $\hat{w}$ be the origin of $\hat{R}_{\hat{u}}$ (i.e., $\hat{R}_{\hat{u}} = A_{\hat{w}}$). Since $\hat{u}$ is a quad-I period of $\hat{R}$, it is also a quad-I period of $\hat{R}_{\hat{u}}$. By Lemma 3, $\hat{v} - \hat{u}$ is a quad-II period of $\hat{R}_{\hat{u}}$. Since $\mathit{size}(\hat{R}_{\hat{u}}) = (p - r_1, q - c_1)$, $p - r_1 \geq \lceil m/2\rceil$, and $q - c_1 \geq c_1 + (c_2 - c_1) + \lceil m/2\rceil$, $\hat{R}_{\hat{u}} = A_{\hat{w}}$ contains the lower-right $\lceil m/2\rceil \times (c_1 + (c_2 - c_1) + \lceil m/2\rceil)$ rectangle of A. Let $b_1 = (x_1, y_1)$ and $b_2 = (x_2, y_2)$ be the basis vectors of $A_{\hat{w}}$. Since $|b_1| \leq |\hat{u}|$ and $|b_2| \leq |\hat{v} - \hat{u}|$, $A_{\hat{w}}$ contains an $\lceil m/2\rceil \times (y_1 + y_2 + \lceil m/2\rceil)$ rectangle.

The parameters u, v of FINDLATTICE are transformed to $u, v - u$ with $|v - u| \leq |v|$. Since we start with two vectors whose lengths are less than $m/4$, we end with two such vectors. Thus $A_{\hat{w}}$ is lattice-generative, and its basis vectors b_1, b_2 satisfy $|b_1|, |b_2| < m/4$.

Consider one call of FINDLATTICE. Let w be the origin of R (i.e., $R = A_w$). Note that if w is a period of A, then since u is a quad-I period of R, $w + u$ (the origin of R_u) is also a period of A by Lemma 1. Since we start FINDLATTICE with A, the origin $\hat{w}$ of $A_{\hat{w}}$ is a period of A. Therefore, $A_{\hat{w}} = A_{-\hat{w}}$. □

COROLLARY 4. *If A is radiant-periodic in quad-*I*, quadrants* I *and* III *are lattice-generative.*

In the following, we will refer to the point found in Lemma 7 as $\hat{w}$.

LEMMA 8. *A nonlattice point $v \in C_{\mathrm{I}}$ ($v \in C_{\mathrm{II}}$) on b_1, b_2 is not a quad-*I *(quad-*II*) period of $A_{\hat{w}}$.*

Proof. We prove the lemma for $v \in C_{\mathrm{I}}$ (the other case is similar). Let $B = A_{\hat{w}}$ and $z = (\lceil m/4\rceil - 1, \lceil m/4\rceil - 1)$. By Lemma 7, B_z contains an $\lceil m/4\rceil \times (y_1 + y_2 + \lceil m/4\rceil)$ rectangle since $\lceil m/2\rceil \geq 2\lceil m/4\rceil - 1$. Since $|b_1|, |b_2| < m/4$, B_z covers the unit cell

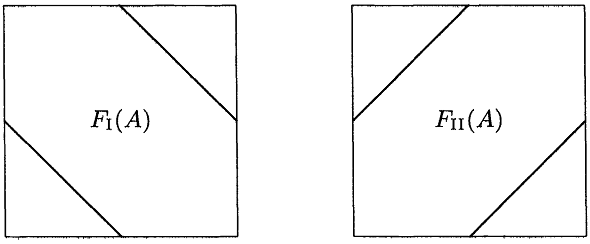


FIG. 5. $F_{\rm I}(A)$ *and* $F_{\rm II}(A)$.

on b_1, b_2. By Theorem 1, a nonlattice point $v \in C_{\rm I}$ on b_1, b_2 is not a quad-I period of B. □

COROLLARY 5. *A nonlattice point $v \in C_{\rm I}$ on b_1, b_2 is not a valid quad-*I *period of A.*

LEMMA 9. *If A is radiant-periodic in quad-*I*, there is no quad-*II *period v of A such that $|v| < m/2$.*

Proof. Since A is radiant-periodic in quad-I, there is no valid quad-II period (i.e., $|v| < m/4$). We now show that there is no quad-II period v of A such that $m/4 \leq |v| < m/2$. Suppose there exist such quad-II periods. Let v_2 be the smallest among them. That is, A is lattice-generative with basis vectors $v_{\rm I}, v_2$. By Theorem 1 with A, basis vectors $v_{\rm I}, v_2$, and $z = (\lceil m/4 \rceil - 1, \lceil m/4 \rceil - 1)$, all valid periods $v \in C_{\rm I}$ are lattice points on $v_{\rm I}, v_2$. Since $|v_2| \geq m/4$, there is only one line (passing through $(0,0)$ and $v_{\rm I}$) of lattice points in $C_{\rm I}$. Thus $\hat{v}_{\rm I} \in C_{\rm I}$ is not a lattice point on $v_{\rm I}, v_2$. This is a contradiction. □

DEFINITION. *Let B be a lattice-generative subrectangle of A and v_1, v_2 be the basis vectors of B. If $A[w] \neq A[u]$ for $w \in A$ and $u \in B$ such that w, u are lattice-congruent modulo v_1, v_2, we say that w is a* defect *with respect to the lattice of B.*

For the rest of this section, a defect will mean a defect with respect to the lattice of quadrant I of A (or the lattice of $A_{-\hat{w}}$). In quadrant II or IV of A, there is at least one defect—otherwise, A would be lattice-periodic. Without loss of generality, assume that quadrant II has at least one defect.

DEFINITION. *A vector v is* affected *by a defect u if at least one of $u + v$ and $u - v$ is in A. A vector v is affected by a set of defects if it is affected by at least one of them. Let $F_{\rm I}(A)$ be the set of points on forward diagonals d of A such that $-m/2 < d < m/2$ and $F_{\rm II}(A)$ be the set of points on backward diagonals d such that $m/2 - 1 < d < 3m/2 - 1$. See Fig.* 5.

LEMMA 10. *Let N be a set of defects in quadrant* II*, and x (y) be the smallest row (largest column) that has a defect in N. Then a vector $(r, c) \in C_{\rm I}$ is not affected by N if and only if $r \geq m - x$ and $c > y$. Similarly, let N' be a set of defects in quadrant* IV*, and x' (y') be the largest row (smallest column) that has a defect in N'. A vector $(r, c) \in C_{\rm I}$ is not affected by N' if and only if $r > x'$ and $c \geq m - y'$.*

Proof. We prove the lemma for quadrant II (the proof for quadrant IV is similar). Let $v = (r, c)$. Since $|v| < m/4$, the points u such that none of $u + v$ and $u - v$ are in A are exactly the points in $B = A[m - r..m - 1, 0..c - 1]$. Thus v is not affected by N if and only if the defects in N are in B if and only if $x \geq m - r$ and $y \leq c - 1$. □

LEMMA 11. *If A is radiant-periodic in quad-*I*, all defects are in $A - F_{\rm I}(A)$.*

Proof. By Corollary 5, nonlattice points $v \in C_{\rm I}$ on b_1, b_2 are not quad-I periods of A. That is, $v_{\rm I}$ and $\hat{v}_{\rm I}$ are lattice points on b_1, b_2. Thus if u is a defect, then each of the four points $u + v_{\rm I}, u + \hat{v}_{\rm I}, u - v_{\rm I}, u - \hat{v}_{\rm I}$ which is in A must also be a defect. If

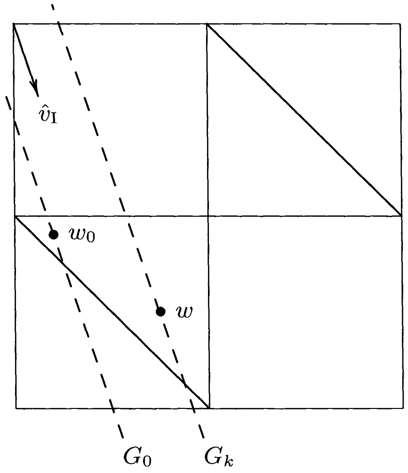


FIG. 6. *G_0 and G_k.*

v_{I} is counterclockwise with respect to $\hat{v}_{\mathrm{I}}$, let $v_1 = v_{\mathrm{I}}$. Otherwise, let $v_1 = -v_{\mathrm{I}}$. In the following, we consider the case $v_1 = v_{\mathrm{I}}$ (the other case is similar).

Suppose there is a defect $w_0 \in F_{\mathrm{I}}(A)$. The point w_0 is in quadrant II or IV by Corollary 4. Without loss of generality, assume that w_0 is in quadrant II. For $i \geq 0$, let G_i be the line passing through $w_0 + iv_1$ and parallel to $\hat{v}_{\mathrm{I}}$ and $size(G_i)$ be the length of the intersection of G_i and A. Let $k \geq 0$ be the largest number such that G_k contains a defect w in quadrant II. Since $w_0 \in G_0$ is in $F_{\mathrm{I}}(A)$ and $|\hat{v}_{\mathrm{I}}| < m/4$, one of $w_0 + \hat{v}_{\mathrm{I}}$ and $w_0 - \hat{v}_{\mathrm{I}}$ (which are in G_0) is in A. Since G_k, G_0 are parallel and G_k is nearer to the center of A than G_0, we have $size(G_k) \geq size(G_0)$ and thus one of $w + \hat{v}_{\mathrm{I}}$ and $w - \hat{v}_{\mathrm{I}}$ is also in A. See Fig. 6. Without loss of generality, assume that $w - \hat{v}_{\mathrm{I}}$ is in A; it must be a defect. If $w - \hat{v}_{\mathrm{I}}$ is in quadrant I, we have a contradiction to Corollary 4. Therefore, assume that $w - \hat{v}_{\mathrm{I}}$ is in quadrant II. Since $|v_{\mathrm{I}}| \leq |\hat{v}_{\mathrm{I}}|$, we have the following two cases.

Case 1: *$v_{\mathrm{I}} - \hat{v}_{\mathrm{I}}$ is a quad-*II *vector.* Since w is in quadrant II, $w + v_{\mathrm{I}} - \hat{v}_{\mathrm{I}}$ is in A. Since $A[w] = A[w - \hat{v}_{\mathrm{I}}] = A[w + v_{\mathrm{I}} - \hat{v}_{\mathrm{I}}]$ by periods v_{I} and $\hat{v}_{\mathrm{I}}$, $w + v_{\mathrm{I}} - \hat{v}_{\mathrm{I}}$ is a defect. Note that $w + v_{\mathrm{I}} - \hat{v}_{\mathrm{I}}$ is in G_{k+1}. Since $w - \hat{v}_{\mathrm{I}}$ is in quadrant II and v_{I} is a quad-I vector, $w + v_{\mathrm{I}} - \hat{v}_{\mathrm{I}}$ is in quadrant II or III. If it is in quadrant II, we have a contradiction to the maximality of k. If it is in quadrant III, we have a contradiction to Corollary 4.

Case 2: *$\hat{v}_{\mathrm{I}} - v_{\mathrm{I}}$ is a quad-*I *vector.* Since $v_{\mathrm{I}}, \hat{v}_{\mathrm{I}} - v_{\mathrm{I}}$ are quad-I vectors, we have $w - \hat{v}_{\mathrm{I}} \prec_{\mathrm{I}} w + v_{\mathrm{I}} - \hat{v}_{\mathrm{I}} \prec_{\mathrm{I}} w$, and therefore $w + v_{\mathrm{I}} - \hat{v}_{\mathrm{I}}$ is in quadrant II. As in Case 1, $w + v_{\mathrm{I}} - \hat{v}_{\mathrm{I}} \in G_{k+1}$ is a defect and we get a contradiction to the maximality of k. □

DEFINITION. *Let x_2 (y_2) be the smallest row (largest column) that has a defect in quadrant* II *and x_4 (y_4) be the largest row (smallest column) that has a defect in quadrant* IV, *if any. Let $w = (\max(m - x_2, x_4 + 1), \max(y_2 + 1, m - y_4))$. Then A_w is the maximum rectangle containing $A_{\hat{w}}$ (also containing quadrant* III) *such that both A_w and A_{-w} are lattice-generative with the basis vectors b_1, b_2. Let D_{I} be the intersection of C_{I} and A_w. See Fig.* 7.

THEOREM 3. *A vector $v \in D_{\mathrm{I}}$ is a valid quad-*I *period if and only if v is a lattice point on b_1, b_2. Further, there may be valid quad-*I *periods outside D_{I}.*

(i) *If $v_{\mathrm{I}} \in D_{\mathrm{I}}$, there are no valid quad-*I *periods outside D_{I}.*

(ii) *If $v_{\mathrm{I}} \notin D_{\mathrm{I}}$, let $s_1, \ldots, s_p$ be the valid quad-*I *periods from shortest to longest*

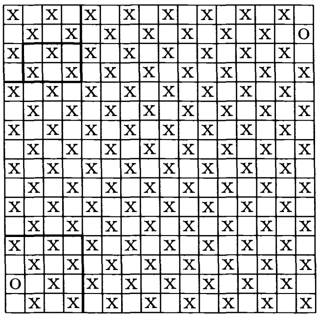


FIG. 7. *Radiant-periodic* (*case* (i)): $b_1 = (1,1), b_2 = (-1,1)$. *Valid quad-*I *periods are* $(2,2)$, $(3,1)$, *and* $(3,3)$. $D_{\rm I}$ *is the marked* 2×3 *rectangle in* $C_{\rm I}$.

which are outside $D_{\rm I}$. *Then* (1) *each* s_i *for* $1 \le i \le p$ *is a lattice point on* b_1, b_2; (2) *the valid periods form a monotone line*; (3) $|s_i - s_{i-1}| \ge |s_{i+1} - s_i|$ *for* $1 \le i \le p-1$, *where* $s_0 = (0,0)$.

Proof. By Corollary 5, nonlattice points $v \in C_{\rm I}$ on b_1, b_2 are not quad-I periods of A. Since A_w and A_{-w} are lattice-generative with the same basis vectors b_1, b_2, all lattice points in $D_{\rm I}$ are valid quad-I periods of A.

For case (i), suppose $v = (r,c)$ is a valid quad-I period outside $D_{\rm I}$. Since $v_{\rm I} = (r_1, c_1)$ is the shortest valid period, we have $r > r_1$ or $c > c_1$. Since v is outside $D_{\rm I}$ and $v_{\rm I}$ is in $D_{\rm I}$, we have $r < r_1$ or $c < c_1$. Hence either $r < r_1$ and $c > c_1$ or $r > r_1$ and $c < c_1$. Without loss of generality, assume $r < r_1$ and $c > c_1$ (i.e., v is above $D_{\rm I}$). Then $v - v_{\rm I}$ is a quad-II period of A_v by Lemma 3. Since $v_{\rm I}$ is a quad-I period of A_v, A_v is lattice-generative.

In the following two paragraphs, we will show that b_1 and b_2 are quad-I and quad-II periods of A_v, respectively. That is, A_v has no defects with respect to the lattice of $A_{\hat{w}}$. Since $A_v = A_{-v}$, there are no defects in rows $r, \ldots, m-1-r$ of A. Since v is above $D_{\rm I}$, we have $r < \max(m - x_2, x_4 + 1)$, i.e., either $m - r > x_2$ or $r \le x_4$. This is a contradiction to the definition of x_2 and x_4.

We first show that $v - v_{\rm I}$ is a lattice point on b_1, b_2. (We already know that $v_{\rm I}$ is.) Since $v \in C_{\rm I}$, A_v contains a $\lceil 3m/4 \rceil \times \lceil 3m/4 \rceil$ square (by $m - \lceil m/4 \rceil + 1 \ge \lceil 3m/4 \rceil$). If A_v contains $A_{\hat{w}}$, $v - v_{\rm I}$ is a lattice point by Lemma 8. Otherwise (the width of $A_{\hat{w}}$ is larger than that of A_v), let R be the lower-right $\lceil m/2 \rceil \times \lceil 3m/4 \rceil$ rectangle of A. Note that R is contained in A_v and also in $A_{\hat{w}}$. By Theorem 1 with R, basis vectors b_1, b_2, and $z = (-\lceil m/4 \rceil + 1, \lceil m/4 \rceil - 1)$, $v - v_{\rm I}$ is a lattice point on b_1, b_2. (Since R_z is at least an $\lceil m/4 \rceil \times \lceil m/2 \rceil$ rectangle, it covers the unit cell on b_1, b_2.)

We now show that b_1 and b_2 are quad-I and quad-II periods of A_v, respectively. We show that b_1 is a quad-I period of A_v (b_2 is similar). Let $B = A_v$ and u be a point in B_{-b_1}. Note that A_v contains quadrant III of A. Since quadrant III contains the unit cell on $v_{\rm I}, v - v_{\rm I}$, there is u_1 (u_2) in quadrant III which is lattice-congruent to u ($u + b_1$) modulo $v_{\rm I}, v - v_{\rm I}$ by Lemma 6. By periods $v_{\rm I}, v - v_{\rm I}$ of A_v, we have $A[u] = A[u_1]$ and $A[u + b_1] = A[u_2]$. Since $v_{\rm I}, v - v_{\rm I}$ are lattice points on b_1, b_2, u_1 is lattice-congruent to u_2 modulo b_1, b_2. By periods b_1, b_2 of quadrant III, we have $A[u_1] = A[u_2]$. Thus $A[u] = A[u + b_1]$.

Case (ii) is proved by the following.

(1) Each s_i is a lattice point on b_1, b_2 because nonlattice points cannot be valid quad-I periods by Corollary 5.

(2) Suppose two valid periods s_i and s_j for $i < j$ create a quad-II vector (i.e.,

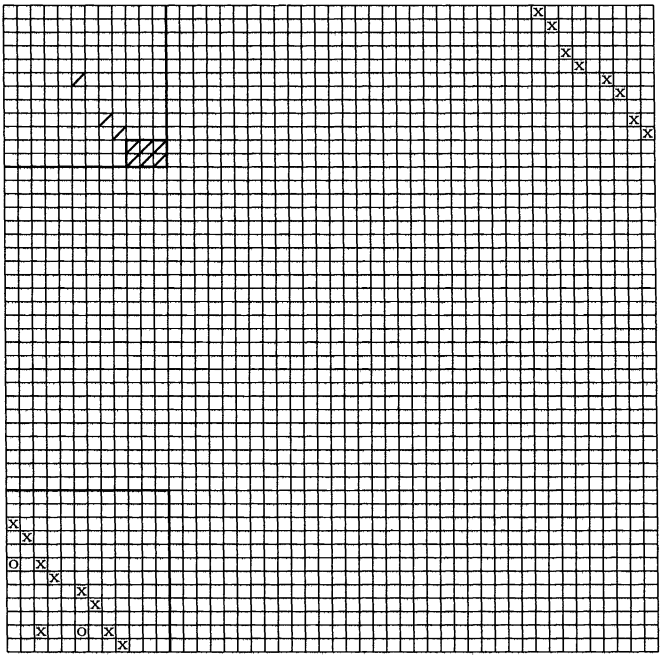


FIG. 8. *Radiant-periodic (case* (ii)): $b_1 = (0,1)$, $b_2 = (-1,0)$. *Valid quad-*I *periods are marked with slashes.* $D_{\rm I}$ *is the marked* 2×3 *rectangle in* $C_{\rm I}$.

either $s_j - s_i$ or $s_i - s_j$ is a quad-II vector). Without loss of generality, assume $s_j - s_i$ is a quad-II vector. Then A_{s_j} is lattice-generative with periods s_i and $s_j - s_i$ by Lemma 3. As in case (i), $s_j - s_i$ is a lattice point on b_1, b_2 and b_1, b_2 are periods of A_{s_j}. We get the same contradiction as in case (i). Therefore, $s_1, \ldots, s_p$ form a monotone line. See Fig. 8.

(3) The proof of (3) is the same as the proof of Theorem 2. □

Even when quadrants I and III of A are lattice-generative, A may be line-periodic if the valid quad-I periods form a line passing through $(0,0)$.

4. The witness computation.

We will compute witnesses of a two-dimensional array $P[0..m-1, 0..m-1]$ in $O(m^2)$ time. Combined with the alphabet-independent text-processing algorithm of [4], this leads to an alphabet-independent linear-time algorithm for two-dimensional pattern matching. In a one-dimensional string, witnesses can be computed from left to right [13, 8]. In a two-dimensional array, computing witnesses from the upper-left and lower-left corners of the array does not work because the computed witnesses may be out of boundary in performing duels. We compute witnesses from the center of the array towards the outside. (We may also start from $(\lceil m/2 \rceil - 1, 0)$.)

DEFINITION. *Let* $\hat{t} = \lceil \log_2 m \rceil$. *We define subsquares* P^t *of the array* P *for* $3 \le t \le \hat{t}$ *recursively:* $P^{\hat{t}} = P$. *Let* $\hat{m}$ *be the length of the sides of* P^t *(i.e.,* P^t *is an* $\hat{m} \times \hat{m}$ *square). Then* $P^{t-1} = P^t[\lfloor \hat{m}/4 \rfloor .. \lfloor \hat{m}/4 \rfloor + \lceil \hat{m}/2 \rceil - 1, \lfloor \hat{m}/4 \rfloor .. \lfloor \hat{m}/4 \rfloor + \lceil \hat{m}/2 \rceil - 1]$. *That is,* P^{t-1} *is the* $\lceil \hat{m}/2 \rceil \times \lceil \hat{m}/2 \rceil$ *square in the middle of* P^t. *Note that* $2^{t-1} < \hat{m} = \lceil m/2^{\hat{t}-t} \rceil \le 2^t$. *The* center v_c *of* P^t *(not necessarily a point in* P^t*) is taken to be* $(i_c, j_c) = (\hat{m}/2 - 0.5, \hat{m}/2 - 0.5)$. *Let* $C^t_{\rm I}$ $(C^t_{\rm II})$ *be the set of all quad-*I *(quad-*II*) vectors* v *such that* $|v| < \hat{m}/4$. *Note that* $C^t_{\rm I}$ $(C^t_{\rm II})$ *contains* $C^{t-1}_{\rm I}$ $(C^{t-1}_{\rm II})$ *even though this does not appear so in Fig.* 9.

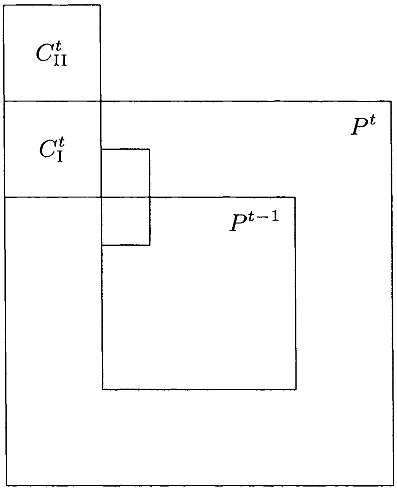


FIG. 9. *Stage t in pattern processing.*

The witness computation has $\hat{t}-2$ stages from $t=3$ to $\hat{t}$. At stage t,

- consider P^t as an entire array (row and column indices are $0,\ldots,\hat{m}-1$);
- compute WITNESS$[0..\lceil\hat{m}/4\rceil-1, 1..\lceil\hat{m}/4\rceil-1]$ (denoted by WITNESS(C^t_{I})) and WITNESS$[-\lceil\hat{m}/4\rceil+1..-1, 0..\lceil\hat{m}/4\rceil-1]$ (denoted by WITNESS(C^t_{II})). WITNESS$[v]=(m,m)$ for $v\in C^t_{\mathrm{I}}$ (for $v\in C^t_{\mathrm{II}}$) if v is a valid quad-I (quad-II) period of P^t; otherwise, WITNESS$[v]=w$ such that $P^t[w]\neq P^t[w-v]$ (i.e., $w, w-v\in P^t$). The points w and $w-v$ are called a *witness* and a *cowitness* of P^t against v, respectively.

Initially ($t=3$), WITNESS(C^3_{I}) and WITNESS(C^3_{II}) are computed by symbol comparisons in constant time. At stage $t>3$, we compute WITNESS(C^t_{I}) and WITNESS(C^t_{II}), given WITNESS(C^{t-1}_{I}) and WITNESS(C^{t-1}_{II}).

DEFINITION. *At stage t we say that v is a* candidate *if it can still be a valid period of P^t. Consider two candidates u and v (both quad-*I *or both quad-*II*) such that $u\prec_{\mathrm{I}} v$ or $u\prec_{\mathrm{II}} v$. Let w be a witness against $v-u$.*

1. *Type-*1 *duel:*

   1a. *If $P^t[w+u]\neq P^t[w]$, $w+u$ is a witness against u.*

   1b. *If $P^t[w+u]\neq P^t[w-v+u]$, $w+u$ is a witness against v.*

2. *Type-*2 *duel:*

   2a. *If $P^t[w-v]\neq P^t[w]$, w is a witness against v.*

   2b. *If $P^t[w-v]\neq P^t[w-v+u]$, $w-v+u$ is a witness against u.*

The point $w+u$ (point $w-v$) will be called the apex *of the type-*1 *(type-*2*) duel. We say that the type-*1 *(type-*2*) duel is* legal *if its apex $w+u$ ($w-v$) is in P^t. The duel between u and v for $u\prec_{\mathrm{I}} v$ (or $u\prec_{\mathrm{II}} v$) will be denoted by $\langle u:v\rangle$.*

Consider a duel $\langle u:v\rangle$. We use one of the type-1 and type-2 duels which is legal and find witnesses against one (or both) of u and v . We will refer below to the parallelogram of Fig. 10 and we summarize its relevant properties.

FACT 2. (1) *If a type-*1 *duel is used, its apex $w+u$ is always the witness against u or v (or both), and w is the cowitness against u, or $w-v+u$ is the cowitness against v (both can happen).*

(2) *If a type-*2 *duel is used, its apex $w-v$ is always the cowitness against u or v (or both), and w is the witness against v, or $w-v+u$ is the witness against u (both can happen).*

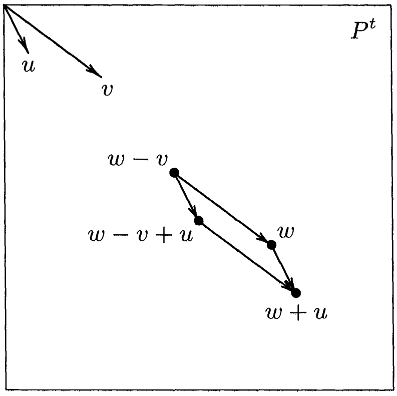


FIG. 10. *Type-1 and type-2 duels.*

Stage t has two major ingredients:

(a) Move WITNESS(C_{I}^{t-1}) into WITNESS(C_{I}^{t}) (WITNESS(C_{II}^{t-1}) into WITNESS(C_{II}^{t})). Although the upper-left corners of $P^3, \ldots, P^{\log m}$ are all different, WITNESS arrays can be carried on through the stages. That is, if WITNESS$[v] = (r, c) \in P^{t-1}$, then WITNESS$[v] = (r + \lfloor \hat{m}/4 \rfloor, c + \lfloor \hat{m}/4 \rfloor) \in P^t$. Note that $(r, c) \in P^{t-1}$ and $(r + \lfloor \hat{m}/4 \rfloor, c + \lfloor \hat{m}/4 \rfloor) \in P^t$ refer to the same point in P. This process will be referred to as the *translation of witnesses*.

(b) Perform duels in C_{I}^{t} and C_{II}^{t} using WITNESS(C_{I}^{t-1}) and WITNESS(C_{II}^{t-1}).

LEMMA 12. *When we perform duels in C_{I}^{t} and C_{II}^{t} using* WITNESS(C_{I}^{t-1}) *and* WITNESS(C_{II}^{t-1}), *both type-1 and type-2 duels are legal.*

Proof. We prove the lemma only for type-1 duels and the case when $v - u \in C_{\mathrm{I}}^{t-1}$. The other cases are similar. Consider a duel $\langle u : v \rangle$ such that $u, v \in C_{\mathrm{I}}^{t}$ and $v - u \in C_{\mathrm{I}}^{t-1}$. Let $w =$ WITNESS$[v - u]$, and let $u = (i, j)$, $v = (k, l)$, and $w = (r, c)$. Since w is in P^{t-1} (i.e., $\lfloor \hat{m}/4 \rfloor \leq r, c \leq \lfloor \hat{m}/4 \rfloor + \lceil \hat{m}/2 \rceil - 1$), we have $\lfloor \hat{m}/4 \rfloor \leq r + i, c + j \leq \lfloor \hat{m}/4 \rfloor + \lceil \hat{m}/4 \rceil + \lceil \hat{m}/2 \rceil - 2 \leq \hat{m} - 1$. Thus $w + u = (r + i, c + j)$ is in P^t.

Consider a duel $\langle u : v \rangle$ such that $u, v \in C_{\mathrm{II}}^{t}$ and $v - u \in C_{\mathrm{I}}^{t-1}$. Let $w =$ WITNESS$[v - u]$, and let $u = (i, j)$, $v = (k, l)$, and $w = (r, c)$. Since $-\hat{m}/4 < i < 0$ and $0 \leq j < \hat{m}/4$, we have $0 \leq r + i < \lfloor \hat{m}/4 \rfloor + \lceil \hat{m}/2 \rceil$ and $\lfloor \hat{m}/4 \rfloor \leq c + j < \hat{m}$, so $w + u = (r + i, c + j)$ is in P^t. □

For a given vector v, let $\mathcal{L}$ be the line passing through $(0, 0)$ and parallel to v. Given v and P^t, we can find the valid periods of P^t among the vectors on $\mathcal{L}$ (and witnesses against the nonperiods among them) in linear time by partitioning P^t into lines parallel to v and checking periodicity in each line. Periodicity of each line is checked by using Algorithm 1 in Main and Lorentz [12] (which is a variation of the KMP algorithm) as follows. For a line consisting of p elements of P^t, we construct a string $S[0..p-1]$, where $S[i]$ is the ith element of the line, and compute $LP[1..p-1]$ in linear time by Algorithm 1 in [12], where $LP[i]$ is the length of the longest prefix of S which starts at position i of S. If $i + LP[i] < p$, the point in P^t corresponding to position $i + LP[i]$ in S is a witness against the vector joining the 0th element and the ith element of the line since $S[LP[i]] \neq S[i + LP[i]]$ by the definition of LP. See Fig. 11 for an example. All the vectors on $\mathcal{L}$ for which a witness is not found after considering all the lines parallel to v are exactly all the valid periods of P^t among the vectors on $\mathcal{L}$. This procedure will be referred to as LINE.

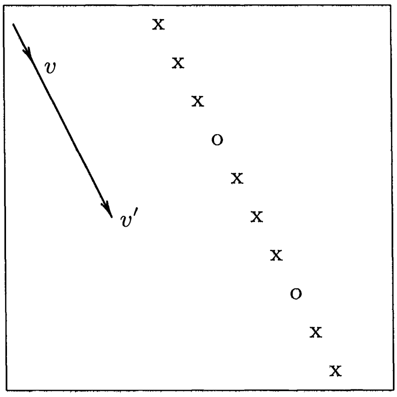


FIG. 11. *A line parallel to v that consists of* 10 *elements.* $LP[5] = 2$ *means* $S[7]$ (*bottom* o) *is a witness against* $v' = 5v$.

We now describe stage $t > 3$. First, we translate WITNESS($C_{\rm I}^{t-1}$) and WITNESS($C_{\rm II}^{t-1}$) into WITNESS($C_{\rm I}^{t}$) and WITNESS($C_{\rm II}^{t}$), respectively. The rest of stage t depends on the category of P^{t-1}.

4.1. P^{t-1} is nonperiodic. Since all of $C_{\rm I}^{t-1}$ and $C_{\rm II}^{t-1}$ have witnesses in this case, we can perform duels between any two candidates in each quadrant of $C_{\rm I}^{t}$. In each of the three quadrants of $C_{\rm I}^{t}$ except $C_{\rm I}^{t-1}$, perform type-1 duels until at most one candidate survives in the quadrant. We can take any order in performing duels in a quadrant. Since at most one candidate survives in each quadrant, at most three candidates survive in $C_{\rm I}^{t}$. Check for each surviving candidate if it is a valid quad-I period of P^t by symbol comparisons. Repeat a similar procedure for $C_{\rm II}^{t}$. P^t can be of any of the four categories.

4.2. P^{t-1} is lattice-periodic. Let $v_{\rm I}$ and $v_{\rm II}$ be the basis vectors of P^{t-1}. In this subsection, a defect will mean a defect with respect to the lattice of P^{t-1}. The following lemma shows transitions of periodicities from P^{t-1} to P^t when P^t has defects.

LEMMA 13. (1) *If there is a defect in P^t, then P^t is not lattice-periodic.*

(2) *If there are defects in two adjacent quadrants of P^t, then P^t is not radiant-periodic.*

Proof. (1) Let v be a defect in P^t. Suppose that P^t is lattice-periodic and v_1, v_2 are the basis vectors of P^t. By Theorem 1 with P^{t-1}, basis vectors $v_{\rm I}, v_{\rm II}$, and $z = (\lceil \hat{m}/4 \rceil - 1, \lceil \hat{m}/4 \rceil - 1)$, v_1 and v_2 must be lattice points on $v_{\rm I}, v_{\rm II}$. Since P^{t-1} covers the unit cell on v_1, v_2, by Lemma 6, there is a point $u \in P^{t-1}$ which is lattice-congruent to v modulo v_1, v_2. Since u and v are lattice-congruent modulo v_1, v_2 and v_1, v_2 are lattice points on $v_{\rm I}, v_{\rm II}$, u and v are lattice-congruent modulo $v_{\rm I}, v_{\rm II}$. Hence we have $P^t[u] \neq P^t[v]$. Therefore, v is a defect with respect to the lattice on v_1 and v_2, which is a contradiction.

(2) Let u and v be defects in quadrants II and III, respectively. The other cases are similar. Suppose P^t is radiant-periodic in quad-I and b_1, b_2 are the basis vectors of quadrant I of P^t. By Lemma 11, there are no defects in $F_{\rm I}(P^t)$ with respect to the lattice on b_1, b_2. Since $F_{\rm I}(P^t)$ contains P^{t-1}, by Theorem 1 with P^{t-1}, basis vectors $v_{\rm I}, v_{\rm II}$, and $z = (\lceil \hat{m}/4 \rceil - 1, \lceil \hat{m}/4 \rceil - 1)$, b_1 and b_2 must be lattice points on $v_{\rm I}, v_{\rm II}$. As in (1), there is $v' \in P^{t-1}$ which is lattice-congruent to v modulo b_1, b_2, and we

have $P^t[v] \neq P^t[v']$ since v and v' are lattice-congruent modulo $v_{\rm I}, v_{\rm II}$. Therefore, v is a defect with respect to the lattice on b_1, b_2. Since v is in quadrant III which is contained in $F_{\rm I}(P^t)$, we have a contradiction with Lemma 11. Similarly, P^t is not radiant-periodic in quad-II because of u. □

FACT 3. *Let v be a lattice point on $v_{\rm I}, v_{\rm II}$.*

(1) *If one of $u, u-v \in P^t$ is a defect and the other is not a defect, u and $u-v$ are a witness and a cowitness against v. In this case, we say that u* provides *a witness against v.*

(2) *If u and $u-v$ are not defects, u cannot be a witness against v.*

DEFINITION. *Let S be a set of points. A vector v is* compatible *with S if for each defect $u \in S$,*

(a) *if $u+v$ is in P^t, then $P^t[u] = P^t[u+v]$, and*

(b) *if $u-v$ is in P^t, then $P^t[u] = P^t[u-v]$.*

The algorithm of the lattice-periodic case has five steps:

A1. Compute witnesses against nonlattice points of $C^t_{\rm I}$ and $C^t_{\rm II}$. These witnesses are computed from the witnesses against nonlattice points of $C^{t-1}_{\rm I}$ and $C^{t-1}_{\rm II}$ which were computed in previous stages.

The remaining steps consider the lattice points of $C^t_{\rm I}$ and $C^t_{\rm II}$.

A2. Compare each element of P^t with the lattice of P^{t-1}. If there are no defects, stop; P^t is lattice-periodic.

A3. If there are defects, check by procedure LINE whether P^t has valid vertical or horizontal periods compatible with the defects. If so, compute witnesses against nonperiods and stop; P^t is line-periodic.

A4. If P^t has no valid vertical or horizontal periods, compute one of WITNESS $(C^t_{\rm I})$ and WITNESS$(C^t_{\rm II})$. In this case, the nearest defect to the center of P^t provides witnesses either for all $C^t_{\rm I}$ or for all $C^t_{\rm II}$.

A5. Compute the other of WITNESS$(C^t_{\rm I})$ and WITNESS$(C^t_{\rm II})$. This step is most involved and it is divided into two cases depending on whether P^t has defects in two adjacent quadrants or not. In each case, we first reduce the number of candidates to $O(\hat{m})$ by computing easy-to-find witnesses. Then among the remaining $O(\hat{m})$ candidates, we find the periods which are compatible with all the defects in P^t.

A5.1. No two adjacent quadrants have defects (i.e., defects are in quadrants I and III or they are in quadrants II and IV). Figure 8 shows an example of this case. By (1) of Lemma 13, P^t can be radiant-periodic (as in Fig. 8), line-periodic, or nonperiodic.

A5.2. Two adjacent quadrants have defects. By Lemma 13, P^t is either line-periodic or nonperiodic.

We now describe the five steps in detail.

Step A1. We compute witnesses against nonlattice points of $C^t_{\rm I}$ and $C^t_{\rm II}$ using the witnesses against nonlattice points of $C^{t-1}_{\rm I}$ and $C^{t-1}_{\rm II}$ as follows. For each nonlattice point v in $C^t_{\rm I} - C^{t-1}_{\rm I}$, there is a point u in $C^{t-1}_{\rm I} \cup C^{t-1}_{\rm II}$ which is lattice-congruent to v modulo $v_{\rm I}, v_{\rm II}$ by Corollary 2 applied to P^{t-1}. Let w be the (computed) witness of P^{t-1} against u (i.e., $P^{t-1}[w] \neq P^{t-1}[w-u]$). Then there is a point w' in quadrant III of P^{t-1} which is lattice-congruent to w by Lemma 6 because quadrant III is a $\lceil \hat{m}/4 \rceil \times \lceil \hat{m}/4 \rceil$ square and $|v_{\rm I}|, |v_{\rm II}| < \hat{m}/8$. Since $v \in C^t_{\rm I}$, $w'-v$ is in P^{t-1}. Since w' $(w-u)$ is lattice-congruent to w $(w'-v)$, we have $P^{t-1}[w'] = P^{t-1}[w] \neq P^{t-1}[w-u] = P^{t-1}[w'-v]$ and therefore w' is a witness against v. Similarly, we compute witnesses against nonlattice points of $C^t_{\rm II}$.

Step A2. We now consider the lattice points of $C^t_{\rm I}$ and $C^t_{\rm II}$. Compare each element of P^t with the lattice of P^{t-1}. If there are no defects, P^t is lattice-periodic, and all lattice points of C^t_{I} (C^t_{II}) are valid quad-I (quad-II) periods of P^t by Fact 3.

Step A3. If there are defects in P^t, let u_c be the nearest defect to the center of P^t in the remaining steps. In this step, we check if P^t has valid vertical or horizontal periods compatible with the defects by procedure LINE, which also computes all witnesses against vertical and horizontal nonperiods. Assume that there are valid vertical or horizontal periods (but not both by (1) of Lemma 13). Without loss of generality, assume there are valid vertical periods (the other case is similar). Since valid vertical periods imply that defects make vertical lines each of which goes through two adjacent quadrants, P^t is line-periodic, but it is neither lattice-periodic nor radiant-periodic by Lemma 13. Thus the valid vertical periods are all the valid periods of P^t. We compute witnesses against other vectors of C^t_{I} and C^t_{II} as follows, and we are done. Since defects make vertical lines and there are no defects in P^{t-1}, the defect u_c is on a column c such that $c < \lfloor \hat{m}/4 \rfloor$ and $c \geq \lfloor \hat{m}/4 \rfloor + \lceil \hat{m}/2 \rceil$ and it is on a row r such that $\lfloor \hat{m}/4 \rfloor \leq r < \lfloor \hat{m}/4 \rfloor + \lceil \hat{m}/2 \rceil$. For all nonvertical and nonhorizontal vectors v of C^t_{I} and C^t_{II}, either $u_c + v$ or $u_c - v$ is not a defect in P^t since $|v| < \hat{m}/4$. Since u_c is a defect and either $u_c + v$ or $u_c - v$ is not, u_c provides witnesses against all nonvertical and nonhorizontal vectors.

Step A4. Assume next that there are no valid vertical or horizontal periods. We compute one of WITNESS(C^t_{I}) and WITNESS(C^t_{II}) as follows. In this case, the nearest defect u_c to the center of P^t provides witnesses either for all C^t_{I} or for all C^t_{II}. If u_c is in quadrant II or IV of P^t (say in quadrant II), we find witnesses against all nonvertical vectors of C^t_{II}. (Note that witnesses against all vertical vectors of C^t_{II} have been computed in Step A3.) The point $u_c + v$ for nonvertical vector $v \in C^t_{\mathrm{II}}$ is closer to the center than u_c because u_c is in quadrant II and v is a quad-II vector. Thus $u_c + v$ is not a defect. Since u_c is a defect and $u_c + v$ is not, u_c provides a witness against v. Computing WITNESS(C^t_{I}) is more involved, and it will be described in Step A5. The case in which u_c is in quadrant I or III is similar. We first compute WITNESS(C^t_{I}) as above and then WITNESS(C^t_{II}) in an analogous way to the one described below.

Step A5. We now explain how to compute WITNESS(C^t_{I}), assuming u_c is in quadrant II or IV. This computation is divided into two cases A5.1 and A5.2 depending on whether P^t has defects in two adjacent quadrants or not.

Case A5.1. First, assume that no two adjacent quadrants have defects (i.e., no defects in quadrants I and III). Then P^t can be radiant-periodic in quad-I, line-periodic in quad-I, or nonperiodic. We will find all valid quad-I periods of P^t compatible with the defects and compute witnesses against the other vectors in C^t_{I}.

It is easy to see that a lattice point $v \in C^t_{\mathrm{I}}$ is a valid period of P^t if and only if v is compatible with quadrants II and IV. Let V be the set of lattice points in C^t_{I}. We will describe below how to find the set $V_{\mathrm{II}} \subset V$ of vectors that are compatible with quadrant II. Similarly, we find the set $V_{\mathrm{IV}} \subset V$ of vectors that are compatible with quadrant IV. The intersection of V_{II} and V_{IV} is the set of valid quad-I periods of P^t. In the course of this computation, the witnesses against nonperiods will be computed.

We describe how to find the vectors of V which are compatible with quadrant II and how to compute witnesses against the other vectors. Let h (h') be the smallest row (largest column) that has a defect in quadrant II. See Fig. 12. For each row $h \leq i < \hat{m}$, let d_i be the largest number such that (i, d_i) is a defect in quadrant II; $d_i = -1$ if row i does not have a defect in quadrant II. Also, $d_i = -1$ for $0 \leq i < h$. Let $u_i = (i, d_i)$. If $d_i = -1$, u_i will be denoted by $*$.

All vectors of V which are not affected by the defects of quadrant II are obviously compatible with quadrant II. Among the vectors (r, c) of V which are affected by the

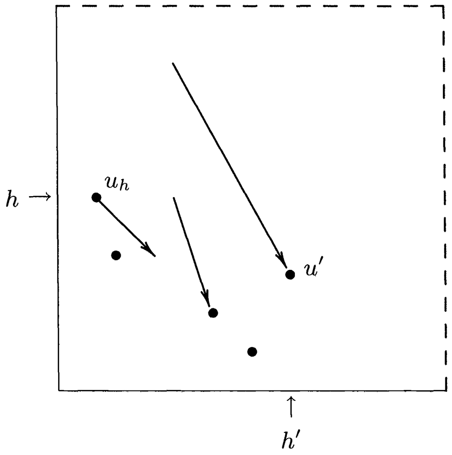


FIG. 12. *Lower-left part of P^t (●'s are defects).*

defects of quadrant II (i.e., $r < \hat{m} - h$ or $c \leq h'$ by Lemma 10), we first compute easy-to-find witnesses against vectors except $u_{h+i} - u_h$ for $0 < i < \hat{m} - h$ by the following procedure. Hence the number of remaining candidates will be $O(\hat{m})$. Let $u' = (r', h')$ be a defect in column h'.

PROCEDURE EASY-WITNESS

1. Consider $v = (r, c)$ such that $r \geq \hat{m} - h$ and $c \leq h'$. Since $c \leq h'$ and u' is in column h', $u' - v$ is in P^t. Since $r' < \hat{m}$ and $r \geq \hat{m} - h$, we have $r' - r < \hat{m} - r \leq h$. Hence $u' - v$ is not a defect, and u' is a witness against v. See the rightmost vector in Fig. 12.

2. Consider $v = (r, c)$ such that $r < \hat{m} - h$ and $v \neq u_{h+r} - u_h$ (i.e., $c \neq d_{h+r} - d_h$).

2.1. Case $c > d_{h+r} - d_h$: Since $d_h + c > d_{h+r}$, $u_h + v$ is not a defect, and it is a witness against v. See the leftmost vector in Fig. 12.

2.2. Case $c < d_{h+r} - d_h$: Since $d_{h+r} - c > d_h$, $u_{h+r} - v$ is not a defect, and u_{h+r} is a witness against v. See the middle vector in Fig. 12.

We now show how to find periods among the remaining $O(\hat{m})$ candidates, i.e., those of the form $u_{h+i} - u_h$. For $0 < i < \hat{m} - h$, we define $v_i = u_{h+i} - u_h$ if $u_h \prec_{\text{I}} u_{h+i}$ (i.e., v_i is a quad-I vector); v_i is undefined otherwise. We define a relation on rows: $i \sim j$ for $i < j$ if $v_{j-i} = (x, y)$ is defined and

(i) $u_i = u_j = *$, or
(ii) $u_i = *$ and $u_j \neq *$ and $d_j < y$, or
(iii) $u_j - u_i = v_{j-i}$ and $P^t[i, 0..d_i]$ is a suffix of $P^t[j, 0..d_j]$.

Note that v_{j-i} (if it is defined) is the only one among the remaining vectors that can map row i to row j. The relation $i \sim j$ means that v_{j-i} is "fine" with respect to rows i and j: in (i), there are no defects in rows i and j; in (ii), for every defect u in row j, $u - v_{j-i}$ is not in P^t; in (iii), v_{j-i} successfully maps defects in row i to defects in row j. We extend the relation $\sim$ to intervals: $[i..i + l] \sim [j..j + l]$ if $i + k \sim j + k$ for $0 \leq k \leq l$.

Note that v_i is compatible with quadrant II if and only if $[h - i..\hat{m} - i - 1] \sim [h..\hat{m} - 1]$. Checking (i) and (ii) takes constant time. Consider all strings involved in (iii) (i.e., $P^t[i, 0..d_i]$ for $d_i \geq 0$) and reverse them. Let *row-string* i be the reverse of $P^t[i, 0..d_i]$. In $O(\hat{m}^2)$ time, we will build a matrix PREFIX such that PREFIX$(i, j) = 1$ if and only if row-string i is a prefix of row-string j. Then (iii) is executed in constant time by checking if $u_j - u_i = v_{j-i}$ and $\text{PREFIX}(i, j) = 1$. As a result, the naïve algorithm for checking whether $[h - i..\hat{m} - i - 1] \sim [h..\hat{m} - 1]$ for every $0 \leq i < \hat{m} - h$ takes $O(\hat{m}^2)$ time.

We also need to find witnesses against nonperiods v_i. If $j - i \not\sim j$ for some $h \leq j < \hat{m}$ when we check $[h - i..\hat{m} - i - 1] \sim [h..\hat{m} - 1]$, we find a witness against $v_i = (x, y)$ as follows.

1. $u_{j-i} \neq *$ and $u_j = *$: since $u_{j-i} + v_i$ is not a defect, u_{j-i} provides a witness against v_i.
2. $u_{j-i} = *$, $u_j \neq *$, and $d_j \geq y$: since $u_j - v_i$ is in P^t and is not a defect, u_j provides a witness against v_i.
3. $u_{j-i} \neq *$, $u_j \neq *$, and $u_j - u_{j-i} \neq v_i$: since $u_j - u_{j-i} \neq v_i$, one of $u_{j-i} + v_i$ and $u_j - v_i$ is not a defect. Hence one of u_{j-i} and u_j provides a witness against v_i.
4. $u_j - u_{j-i} = v_i$ and row-string $j - i$ is not a prefix of row-string j (i.e., $P^t[j - i, 0..d_{j-i}]$ is not a suffix of $P^t[j, 0..d_j]$): any point u in row-string j such that $P^t[u - v_i] \neq P^t[u]$ is a witness against v_i.

For case 4, when $\text{PREFIX}(i, j) = 0$, we need to compute a point u in row-string j which does not match the corresponding point in row-string i.

We now compute the matrix PREFIX. We build a *prefix tree* which is a trie with all row-strings involved in (iii) and in which the children of a node are maintained by a linked list. A node v is *marked* with i if the string from the root to v is row-string i. At stage i, we enter row-string i if it is not empty. Assume that we enter row-string i. We walk down the prefix tree from the root with row-string i. On a node, we check its children by following its linked list. We either find a matching child to continue with or there is no matching child with row-string i. In the latter case, we create a set of nodes corresponding to the rest of row-string i.

We analyze the time to construct the prefix tree. When we walk down the prefix tree to enter row-string i, the matches and creating new nodes are charged to row-string i. A mismatch means that there is a subtree containing at least one leaf (marked with k) which we do not go into. The mismatch is charged to the entry $\text{PREFIX}(k, i)$; the total for mismatches is also $O(\hat{m})$. Thus the total time for building the prefix tree is $O(\hat{m}^2)$.

We now make the prefix tree a compacted trie, i.e., every nonmarked node which has only one child is removed. (Note that we can also make the prefix tree compact while we construct it.) Since all leaves are marked nodes and there are $O(\hat{m})$ marked nodes (including internal nodes having one child), the total number of nodes in the compact prefix tree is $O(\hat{m})$. As we traverse the tree from the leaves to the root, for each node v, we compute the list $M(v)$ of row-strings whose corresponding marked nodes are in the subtree rooted at v. Now we traverse the tree and compute PREFIX as follows. On a node v marked with i, we set $\text{PREFIX}(i, j) = 1$ for each $j \in M(v)$. When we traverse the arc from v to its child u, we set $\text{PREFIX}(i, j) = 0$ for each $i \in M(v) - M(u)$ and each $j \in M(u)$. Let k be the number of symbols from the root to v. For every entry $\text{PREFIX}(i, j) = 0$ computed above, the $(k + 1)$st position in row-string j is a mismatch between row-strings i and j. Since each entry of PREFIX is computed once, this computation takes $O(\hat{m}^2)$ time. Traversing the tree also takes $O(\hat{m}^2)$ time as in the construction of the prefix tree.

Case A5.2. We now consider the remaining case and assume that two adjacent quadrants have defects (i.e., there are defects in quadrant I or III in addition to one in quadrant II or IV). Then P^t is either line-periodic or nonperiodic by Lemma 13.

Recall that h (h') is the smallest row (largest column) that has a defect in quadrant II, and d_i is the largest number such that (i, d_i) is a defect in quadrant II. Let $L = \{u_i = (i, d_i) \mid h \leq i < \hat{m}\}$. Let g be the smallest column that has a defect in quadrant IV. For each column $g \leq j < \hat{m}$, let d_j be the largest number such that (d_j, j) is a defect in quadrant IV; $d_j = -1$ if column j does not have a defect in quadrant IV. Let $L' = \{(d_j, j) \mid g \leq j < \hat{m}\}$. Let V be the set of lattice points in C^t_{I}.

LEMMA 14. *If there is a vector v compatible with L (L') such that for some defect $u \in L$ ($u \in L'$), $u + v$ is in quadrant* III *or $u - v$ is in quadrant* I, *then all the vectors of V compatible with L (L') are on the line passing through* $(0,0)$ *and v.*

Proof. We prove the lemma for L (the case for L' is analogous). Let $v = (r, c)$. Without loss of generality, assume that there is $w_0 \in L$ such that $w_0 - v$ is in quadrant I (the other case is similar). Suppose there is a vector $v' \in V$ which is compatible with L and is not on the line passing through $(0,0)$ and v. Without loss of generality, assume that v is counterclockwise with respect to v' (the other case is similar). For $i \geq 0$, let G_i be the line passing through $w_0 + iv$ and parallel to v'. Note that G_{i+1} is to the right of G_i.

Let $k \geq 0$ be the largest number such that G_k contains a defect in L, and let $w = (x, y) \in L$ be the defect in G_k whose row number is the smallest. Since v is counterclockwise with respect to v' and $w_0 - v$ is in quadrant I, G_0 intersects both quadrants I and II. Since $k \geq 0$, G_k also intersects both quadrants I and II. By the minimality of x, the row number of $w - v'$ is less than $\lfloor \hat{m}/2 \rfloor$. Since $|v'| < \hat{m}/4$, we have $x < \hat{m}/4 + \lfloor \hat{m}/2 \rfloor$. Since there are no defects in P^{t-1}, $y < \lfloor \hat{m}/4 \rfloor$. Thus $w + v$ is in quadrant II, and it is a defect by the compatibility of v with L. Moreover, $w + v$ is in L because otherwise there would exist $w' = (x', y') \in L$ such that $x' = x + r$ and $y' > y + c$ and $w' - v$ would be a defect in quadrant II satisfying $x' - r = x$ and $y' - c > y$, so w would not be in L. Since $w + v$ is in G_{k+1}, we have a contradiction to the maximality of k. □

For a vector of V to be a period of P^t, it must be compatible with L and L' by the definition of compatibility. In the following, we describe how to find the vectors compatible with L. Finding the vectors compatible with L' is analogous. If u_h satisfies $h \geq \lfloor \hat{m}/4 \rfloor + \lceil \hat{m}/2 \rceil$ and $d_h \geq \lfloor \hat{m}/4 \rfloor$, then u_h is a witness against all $v \in V$ because $u_h - v$ (which is in quadrant II) is not a defect. Otherwise, $d_h < \lfloor \hat{m}/4 \rfloor$ because there are no defects in P^{t-1}.

Among the vectors (r, c) of V affected by L (i.e., $r < \hat{m} - h$ or $c \leq h'$), we compute easy-to-find witnesses against vectors except $u_{h+i} - u_h$ for $0 < i < \hat{m} - h$ as in the procedure EASY-WITNESS. Let $u' = (r', h') \in L$ be a defect in column h'. Note that $r' \geq h$. In each case below, we will show only that the relevant point is in quadrant II, from which it follows that the point is not a defect as in EASY-WITNESS (since its row coordinate is smaller than h in Case 1 and its column coordinate is larger than the corresponding d_i in Case 2).

Case 1: $r \geq \hat{m} - h$ *and* $c \leq h'$. Since $r < \hat{m}/4$, we have $h > 3\hat{m}/4$. Since $r' \geq h$, $r' > 3\hat{m}/4$. Since $h' - c \geq 0$ and $r' - r > \hat{m}/2$, $u' - v$ is in quadrant II.

Case 2: $r < \hat{m} - h$ *and* $v \neq u_{h+r} - u_h$ (*i.e.*, $c \neq d_{h+r} - d_h$).

Case 2.1: $c > d_{h+r} - d_h$. Since $d_h < \lfloor \hat{m}/4 \rfloor$, $u_h + v$ is in quadrant II.

Case 2.2: $c < d_{h+r} - d_h$. Since u_{h+r} and u_h are in quadrant II, so is $u_{h+r} - v$.

Among the vectors $v \in V$ affected by L, the remaining $O(\hat{m})$ candidates are those of the form $u_{h+r} - u_h$. For each vector $v_i = u_{h+i} - u_h$, we consider all $u \in L$ and check compatibility (i.e., (a) and (b) in the definition of compatibility). It takes $O(\hat{m})$ time for each vector v_i and $O(\hat{m}^2)$ time overall. If some $u \in L$ violates (a) or (b)—say (a)—for v_i, then u provides a witness against v_i because $P^t[u] \neq P^t[u + v_i]$. The vectors compatible with L are the surviving vectors plus the vectors of V unaffected by L.

Let $U \subset V$ be the set of vectors compatible with L and L'. Now we find the periodicity of P^t and the remaining witnesses.

1. If U is empty, P^t is nonperiodic, and all witnesses have been found.

2. If U is a set of vectors on a line going through $(0, 0)$, we check if the vectors are valid periods of P^t by procedure LINE, which also computes the remaining witnesses.

3. If U contains at least two independent vectors, the nearest defect in quadrants I and III to the center will provide witnesses against the vectors of U as follows. Without loss of generality, assume that the nearest defect u'_c is in quadrant III (the other case is similar). Since u'_c is outside P^{t-1}, $u'_c - v$ is in quadrant II, III, or IV. We claim that $u'_c - v$ for $v \in U$ is not a defect. If $u'_c - v$ is in quadrant III, it is not a defect since u'_c is the nearest one. Suppose $u'_c - v$ is in quadrant II or IV (say quadrant II) and it is a defect. Let $u \in L$ be the point in the same row as $u'_c - v$ (possibly $u = u'_c - v$). We have $u + v$ in quadrant III, and by Lemma 14, the vectors of U must be on a line going through $(0, 0)$, a contradiction to the fact that U contains two independent vectors. Therefore, $u'_c - v$ is not a defect and u'_c is a witness for all $v \in U$.

4.3. P^{t-1} is line-periodic (quad-I). Assume that P^{t-1} is line-periodic in quad-I. Let v_{I} be the shortest valid quad-I period of P^{t-1}. The algorithm of the line-periodic case has four steps:

B1. Check whether P^{t-1} has quad-II periods $v \in C^t_{\text{II}}$.

B2. If P^{t-1} has no quad-II periods in C^t_{II} (i.e., all vectors in C^t_{II} have witnesses), compute WITNESS(C^t_{I}) using two stages of duels, Steps D1 and D2, and stop.

D1. Perform duels among the vectors in C^t_{I}. The surviving candidates form a monotone line M since all of C^t_{II} have witnesses.

D2. Perform duels among the vectors in the monotone line M from shortest to longest. Since duels are done from shortest to longest, the surviving candidates form a line, which can be checked by procedure LINE.

B3. If P^{t-1} has quad-II periods in C^t_{II}, let $\hat{v}_{\text{II}}$ be the shortest one among them. In this case, we can show that all the quad-II periods of P^{t-1} are lattice points on v_{I} and $\hat{v}_{\text{II}}$. Perform duels among the vectors in C^t_{I} and check whether the surviving candidates are lattice points on v_{I} and $\hat{v}_{\text{II}}$. If all of them are lattice points, the rest is the same as in the lattice-periodic case.

B4. If some vectors in C^t_{I} are nonlattice points, check whether there exists a nonlattice point which is a period of P^{t-1}. If no such nonlattice points exist (i.e., all nonlattice points in C^t_{I} are not periods of P^{t-1}), the remaining vectors are lattice points and the rest is the same as in the lattice-periodic case. If such a nonlattice point exists, we can prove special properties, by which WITNESS(C^t_{I}) can be computed using Steps D1 and D2 and procedure LINE as in Step B2 and WITNESS(C^t_{II}) can be computed by symbol comparisons.

DEFINITION. *Consider a point (i, j) (on backward diagonal $i + j$). Its* distance *to backward diagonal d is defined to be $|(i + j) - d|$. Let Q^1_{I}, Q^2_{I}, Q^3_{I}, and Q^4_{I} (Q^1_{II}, Q^2_{II}, Q^3_{II}, and Q^4_{II}) be quadrants* I, II, III, *and* IV *of C^t_{I} (C^t_{II}), respectively.*

In the line-periodic and radiant-periodic cases, we use both type-1 and type-2 duels. We have two rules for choosing the type.

Rule 1. Choose the type of the duel whose apex is closer to the center of P^t.

Rule 2. For a duel between $u, v \in C^t_{\text{I}}$, choose the type of the duel whose apex is closer (by the definition of distance above) to the backward diagonal $\hat{m} - 1$.

We will use Rule 1 for all the duels in the line-periodic and radiant-periodic cases except for Step D2, where we will use Rule 2.

We now describe the four steps in detail. In the witness computation, we use computed witnesses to compute new witnesses. Although the algorithmic part is quite simple, the main technical difficulty is in proving that the duels we perform are legal.

Step B1. Since P^{t-1} is line-periodic with the shortest valid period v_{I} (i.e., $|v_{\text{I}}| < \hat{m}/8$), initially all vectors in C^{t-1}_{I} except the valid periods of P^{t-1} on the line passing through v_{I} and $(0, 0)$ have witnesses and all vectors in C^{t-1}_{II} have witnesses. However, P^{t-1} may have a quad-II period v such that $\hat{m}/8 \le |v| < \hat{m}/4$. First, we check if P^{t-1} has such a quad-II period as follows.

B1.1. Perform duels in each of the three quadrants of C^t_{II} except C^{t-1}_{II} (i.e., $Q^1_{\text{II}}, Q^3_{\text{II}}, Q^4_{\text{II}}$) using WITNESS($C^{t-1}_{\text{I}}$) and WITNESS($C^{t-1}_{\text{II}}$). The surviving candidates will form a line parallel to v_{I} in each quadrant and thus at most three lines in C^t_{II}. The losers of the duels get witnesses from the duels.

B1.2. For each line L (parallel to v_{I}) of candidates in C^t_{II}, choose the shortest quad-II vector v and check if it is a period of P^{t-1} by symbol comparisons. Any mismatch kills all quad-II vectors in the line L by the line-periodicity of P^{t-1}, as Lemma 15 will show. Thus the shortest surviving vector among the (at most) three chosen ones, if any, will be the shortest quad-II period of P^{t-1}.

LEMMA 15. *Let w be a witness of P^{t-1} against $v \in C^t_{\text{II}}$. If $v' \in C^t_{\text{II}}$ is a quad-*II *vector such that $v' - v$ or $v - v'$ is a valid quad-*I *period of P^{t-1}, one of w and $w + v' - v$ is a witness of P^{t-1} against v'.*

Proof. We prove the lemma for the case in which $v' - v$ is a valid quad-I period (the other case is similar). Note that both w and $w - v$ are in P^{t-1} and $P^t[w] \ne P^t[w - v]$. Consider the parallelogram L whose four corners are w, $w - v'$, $w - v$, $w + v' - v$. Let $w = (r_1, c_1)$, $w - v' = (r_2, c_2)$, $w - v = (r_3, c_3)$, and $w + v' - v = (r_4, c_4)$. Since the two sides of L are quad-I vector $v' - v$ and quad-II vector v', we have $r_1 < r_2 \le r_3$ and $r_1 \le r_4 < r_3$. Since $|v'| < \hat{m}/4$ and $|v' - v| < \hat{m}/8$, we have $c_1 - \hat{m}/4 < c_2$ and $c_4 < c_1 + \hat{m}/8$. Since w and $w - v$ are in P^{t-1}, one (or both) of $w - v'$ and $w + v' - v$ is in P^{t-1}.

If $w - v'$ is in P^{t-1}, then w is a witness against v' because $P^t[w - v'] = P^t[w - v] (\ne P^t[w])$ by the period $v' - v$. Otherwise, $w + v' - v$ is a witness against v' because $P^t[w + v' - v] = P^t[w] (\ne P^t[w - v])$. □

LEMMA 16. *Let $\hat{v}_{\text{II}}$ be the shortest surviving candidate in C^t_{II} in Step* B1.2 (*i.e., shortest quad-*II *period of P^{t-1}*). *Then we have the following:*

(1) *All valid quad-*I *periods of P^{t-1} are of the form iv_{I} for integer i.*

(2) *All the surviving candidates in C^t_{II} are quad-*II *periods of P^{t-1} and also lattice points on $v_{\text{I}}, \hat{v}_{\text{II}}$.*

Proof. By Theorem 1 with P^{t-1}, basis vectors $v_{\text{I}}, \hat{v}_{\text{II}}$, and $z = (\lceil \hat{m}/8 \rceil - 1, \lceil \hat{m}/8 \rceil - 1)$, all valid quad-I periods of P^{t-1} are of the form iv_{I} for integer i. Let v be the shortest vector (chosen in Step B1.2) in a line L of candidates in C^t_{II}. Since all valid quad-I periods of P^{t-1} are of the form iv_{I}, all candidates in L are of the form $v + iv_{\text{I}}$. Thus if v is a period of P^{t-1}, so are all candidates in L by Lemma 5. Therefore, all the surviving candidates in C^t_{II} are quad-II periods of P^{t-1}.

We now show that the surviving candidates in C^t_{II} are lattice points on $v_{\text{I}}, \hat{v}_{\text{II}}$.

- *Case* 1: *$\hat{v}_{\text{II}}$ is in Q^1_{II}.* By Theorem 1 with P^{t-1}, basis vectors $v_{\text{I}}, \hat{v}_{\text{II}}$, and

$z = (-\lceil \hat{m}/4 \rceil + 1, \lceil \hat{m}/4 \rceil - 1)$, all the surviving candidates in C^t_{II} are lattice points.

- *Case* 2: $\hat{v}_{\text{II}}$ *is in* Q^3_{II}. This case is similar to Case 1.
- *Case* 3: $\hat{v}_{\text{II}}$ *is in* Q^4_{II}. Since one of the candidates in Q^4_{II} is $\hat{v}_{\text{II}}$, all of them are of the form $\hat{v}_{\text{II}} + iv_{\text{I}}$, i.e., the candidates in Q^4_{II} are lattice points. By Theorem 1 with P^{t-1}, $v_{\text{I}}, \hat{v}_{\text{II}}$, and $z = (-\lceil \hat{m}/4 \rceil + 1, \lceil \hat{m}/8 \rceil - 1)$, all the candidates in Q^1_{II} are lattice points. Similarly, all the candidates in Q^3_{II} are lattice points. □

LEMMA 17. *Let $\hat{w}$ be the witness against $v \in C^t_{\text{II}}$ after Steps* B1.1 *and* B1.2. *Then there are two cases for the locations of the witness $\hat{w}$ and the cowitness $\hat{w} - v$:*

(1) *both $\hat{w}$ and $\hat{w} - v$ are in P^{t-1}, or*

(2) *one is in quadrant* I *or* III *of P^{t-1} and the other is outside P^{t-1} but in $F_{\text{I}}(P^t)$.*

Proof. If $\hat{w}$ was computed before Step B1.1 (i.e., in stage $\leq t-1$), both $\hat{w}$ and $\hat{w}-v$ are in P^{t-1} because $\hat{w}$ is a witness of P^{t-1} at stage $t-1$. If $\hat{w}$ is computed during Step B1.2, both $\hat{w}$ and $\hat{w}-v$ are again in P^{t-1} because Step B1.2 checks periodicities of P^{t-1} only.

Let $\hat{w}$ be the witness computed during Step B1.1. Let $u \in C^t_{\text{II}}$ be the vector to which v lost in a duel of B1.1. Then $u - v$ or $v - u$ is in C^{t-1}_{I} or C^{t-1}_{II}. Without loss of generality, assume that $v - u$ is in C^{t-1}_{I} (the other cases are similar). Let w be the witness against $v - u$. Since Steps B1.1 and B1.2 compute witnesses against vectors in $C^t_{\text{II}} - C^{t-1}_{\text{II}}$, all witnesses against vectors in C^{t-1}_{I} and C^{t-1}_{II} were computed before Step B1.1. Thus both the witness w and the cowitness $w - v + u$ are in P^{t-1}. Recall that $w + u$ $(w - v)$ is the apex of the type-1 (type-2) duel. By Lemma 12, both types of duels are legal. There are three cases based on the locations of $w + u$ and $w - v$:

1. If both $w + u$ and $w - v$ are in P^{t-1}, both $\hat{w}$ and $\hat{w} - v$ (which are two among $w, w - v + u, w + u, w - v$ by Fact 2) are in P^{t-1}.
2. If one of $w + u$ and $w - v$ (say $w - v$) is in P^{t-1} and the other is not, Rule 1 chooses the type-2 duel because $w - v$ is closer to the center than $w + u$, so both $\hat{w} = w$ and $\hat{w} - v = w - v$ are in P^{t-1}.
3. If both $w + u$ and $w - v$ are outside P^{t-1}, both w and $w - v + u$ are in quadrant I or quadrant III of P^{t-1} because $w, w - v + u \in P^{t-1}$ and $u, v \in C^t_{\text{II}}$. Without loss of generality, assume that $w, w - v + u$ are in quadrant I of P^{t-1}. Both $w + u$ and $w - v$ cannot be outside $F_{\text{I}}(P^t)$ because $w \in F_{\text{I}}(P^t)$ and $|u|, |v| < \hat{m}/4$. If one of $w + u$ and $w - v$ (say $w + u$) is outside $F_{\text{I}}(P^t)$, we will show below that $w - v$ is closer to the center than $w + u$. Thus the type-2 duel is chosen by Rule 1; $\hat{w} = w$ is in quadrant I of P^{t-1} and $\hat{w} - v = w - v$ is outside P^{t-1} but in $F_{\text{I}}(P^t)$.

Let $w = (x, y)$, $w + u = (i_1, j_1)$, $w - v = (i_2, j_2)$, and $v_c = (i_c, j_c)$, the center of P^t. We now show that $|(w + u) - v_c| > 3\hat{m}/4 - y - 0.5 > |(w - v) - v_c|$.

- Since w is in quadrant I, $u \in C^t_{\text{II}}$ is a quad-II vector, and $w + u$ is outside P^{t-1}, we have $i_c - i_1 \geq \hat{m}/4 > |j_1 - j_c|$. Hence $|(w + u) - v_c| = i_c - i_1$.
- Since $j_1 - i_1 \geq \hat{m}/2$ (i.e., $w + u$ is outside $F_{\text{I}}(P^t)$), we have $3\hat{m}/4 - j_1 \leq \hat{m}/4 - i_1$. Since $j_1 - y$ is the column coordinate of u, it is less than $\hat{m}/4$. Hence $3\hat{m}/4 - y - 0.5 = (j_1 - y) + (3\hat{m}/4 - j_1) - 0.5 < \hat{m}/4 + (\hat{m}/4 - i_1) - 0.5 = i_c - i_1$.
- Since w is in quadrant I, $v \in C^t_{\text{II}}$ is a quad-II vector, and $w - v$ is outside P^{t-1}, we have $j_c - j_2 \geq \hat{m}/4 > |i_2 - i_c|$. Hence $|(w - v) - v_c| = j_c - j_2$.
- Since $y - j_2$ is the column coordinate of v, it is less than $\hat{m}/4$. Hence $j_c - j_2 = (y - j_2) + (j_c - y) < \hat{m}/4 + (j_c - y) = 3\hat{m}/4 - y - 0.5$. □

LEMMA 18. *If there are no surviving candidates in C^t_{II}, then P^t is not radiant-periodic in quad-*I.

Proof. Suppose P^t is radiant-periodic in quad-I. By Corollary 4, quadrant I (also

III) of P^t is lattice-generative with basis vectors b_1 and b_2 satisfying $|b_1|, |b_2| < \hat{m}/4$. By Lemma 11, there are no defects in $F_{\mathrm{I}}(P^t)$ with respect to the lattice of quadrant I, and therefore P^{t-1} is also lattice-generative with basis vectors b_1 and b_2. ($\hat{m}/8 \leq |b_2| < \hat{m}/4$ because P^{t-1} has no valid quad-II period.) Since there are no surviving candidates in C^t_{II}, there exists a witness w of P^t against b_2, i.e., $P^t[w] \neq P^t[w - b_2]$. By Lemma 17, w and $w - b_2$ are in $F_{\mathrm{I}}(P^t)$, a contradiction to Lemma 11. □

The computation after Step B1 depends on whether or not there are surviving candidates in C^t_{II}.

Step B2. First, assume that there are no surviving candidates in C^t_{II} (i.e., all of C^t_{II} have witnesses). Then P^t is not lattice-periodic because there are no valid quad-II periods. By Lemma 18, P^t can be either line-periodic in quad-I or nonperiodic. In order to compute WITNESS(C^t_{I}), we will have two stages of duels: Steps D1 and D2. Step D1 performs duels among the points of C^t_{I} as follows.

Step D1.

D1.1. Perform duels in each of the three quadrants of C^t_{I} except C^{t-1}_{I} (i.e., $Q^2_{\mathrm{I}}, Q^3_{\mathrm{I}}, Q^4_{\mathrm{I}}$) using WITNESS($C^{t-1}_{\mathrm{I}}$) and WITNESS($C^{t-1}_{\mathrm{II}}$). The surviving candidates will form a line in each quadrant (at most four lines in C^t_{I} including the one in C^{t-1}_{I}).

D1.2. Perform further duels between the lines using WITNESS(C^t_{II}) in the process of combining the four quadrants into C^t_{I}. The surviving candidates form a monotone line M because all of C^t_{II} have witnesses.

LEMMA 19. *Let $\hat{w}$ be the witness against $v \in C^t_{\mathrm{I}}$ after Step* D1.1. *Then there are two cases for the locations of the witness $\hat{w}$ and the cowitness $\hat{w} - v$:*

(1) *both $\hat{w}$ and $\hat{w} - v$ are in P^{t-1}, or*

(2) *one is in quadrant* II *or* IV *of P^{t-1} and the other is outside P^{t-1} but in $F_{\mathrm{II}}(P^t)$.*

Proof. This proof is similar to the proof of Lemma 17 because Step D1.1 is analogous to Step B1.1 above. □

LEMMA 20. *Let $\hat{w}$ be the witness against $v \in C^t_{\mathrm{I}}$ after Step* D1. *Then at least one of the witness $\hat{w}$ and the cowitness $\hat{w} - v$ is in P^{t-1}.*

Proof. If v gets the witness w in Step D1.1, one of $\hat{w}$ and $\hat{w} - v$ is in P^{t-1} by Lemma 19. If v gets the witness w in Step D1.2, let $u \in C^t_{\mathrm{I}}$ be the vector to which v lost in a duel. Then $u - v$ or $v - u$ is in C^t_{II} because we use only WITNESS(C^t_{II}) in Step D1.2. Without loss of generality, assume that $v - u$ is in C^t_{II}. Let w be the witness against $v - u$. Since $v - u$ is in C^t_{II}, w and $w - v + u$ satisfy Lemma 17, and we have two cases:

1. w and $w - v + u$ are in P^{t-1}. Since $|u|, |v| < \hat{m}/4$, both $w + u$ and $w - v$ are in P^t (i.e., both types of duels are legal). At least one of $\hat{w}$ and $\hat{w} - v$ (which is one of w and $w - v + u$ by Fact 2) is in P^{t-1}.

2. One of w and $w - v + u$ is in quadrant I or III of P^{t-1} and the other is outside P^{t-1}. Assume that w is in quadrant I (the other cases are similar). Since w is in quadrant I and $u \in C^t_{\mathrm{I}}$, the apex $w + u$ of the type-1 duel is in P^{t-1}. If the type-1 duel is chosen by Rule 1, by Fact 2, the new witness $\hat{w}$ is $w + u$, which is in P^{t-1}. If the type-2 duel is chosen, its apex $w - v$ is closer to the center than $w + u$ by Rule 1, and therefore it is in P^{t-1}. By Fact 2, the new cowitness $\hat{w} - v$ is $w - v$, which is in P^{t-1}. □

Step D2. We perform duels among the points in the monotone line M. Let $v_1, \ldots, v_p$ be the candidates in M from shortest to longest. Step D2 consists of p iterations. At the beginning of iteration $q \geq 1$, we maintain the invariant that $\hat{V}$ is a subset of $\{v_1, \ldots, v_{q-1}\}$ such that

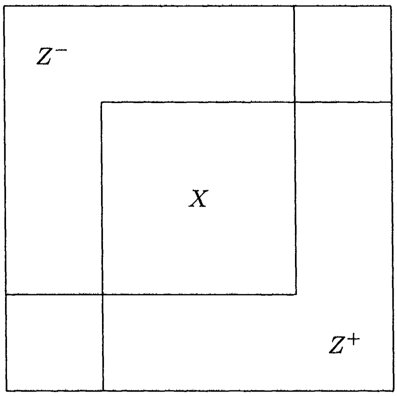


FIG. 13. $X^+ = X \cup Z^+$ *and* $X^- = X \cup Z^-$.

1. either $\hat{V}$ is empty or
2. all the vectors in $\hat{V}$ are on a line $\mathcal{L}$ going through $(0,0)$.

Initially, $\hat{V}$ is empty. At iteration $q \geq 1$, we consider the current candidate v_q and do the following. If $\hat{V}$ is empty or v_q is on the line $\mathcal{L}$, we put v_q into $\hat{V}$. Otherwise, the vector $v_q - u$ for each $u \in \hat{V}$ has a witness because $v_q - u$ is shorter than v_q and it is not on the line $\mathcal{L}$. We perform duels between v_q and the vectors in $\hat{V}$ (in any order among the vectors in $\hat{V}$) until either v_q or all vectors in $\hat{V}$ are removed. The new $\hat{V}$ is the set of surviving vectors among $\hat{V}$ and v_q. Note that the new $\hat{V}$ satisfies the invariant. The losers of the duels get witnesses. Step D2 returns $\hat{V}$ after considering all candidates (thus the remaining vectors in $\hat{V}$ are on one line going through $(0,0)$). We use the procedure LINE to find the valid periods among the vectors in $\hat{V}$ and to compute witnesses for the rest.

We now show that all the duels in Step D2 are legal. Let z and z' be the upper-left and lower-right corners of P^{t-1}, respectively. Let X be the set of points in P^{t-1} and X^+ (X^-) be the set of points in P^t_z $(P^t_{-z'})$. Let $Z^+ = X^+ - X$ and $Z^- = X^- - X$. See Fig. 13.

At the beginning of iteration $q \geq 1$, we maintain the following invariant (called Invariant q): for each vector $v \prec_{\mathrm{I}} v_q$ against which a witness w has been computed, the witness w and the co-witness $w - v$ satisfy the following:

Condition (i). $w, w - v \in X^+$ and $w - v_q \in X^-$; or

Condition (ii). $w, w - v \in X^-$ and $(w - v) + v_q \in X^+$; or

Condition (iii). $w \in Z^+$ and $w - v \in Z^-$.

LEMMA 21. *If* $w \in X^+(Z^+)$, $w - v \in X^-(Z^-)$, *and* $v \prec_{\mathrm{I}} v'$ *for* $v' \in C^t_{\mathrm{I}}$, *then* $w - v'$ *is in* $X^-(Z^-)$ *and* $(w - v) + v'$ *is in* $X^+(Z^+)$.

Proof. We prove the lemma for X^+ and X^- (the case for Z^+ and Z^- is similar). Since $w \in X^+$ and $v' \in C^t_{\mathrm{I}}$, $w - v'$ is in P^t. Since $w - v \in X^-$ and $v \prec_{\mathrm{I}} v'$, $w - v'$ is in X^-. Similarly, $(w - v) + v'$ is in X^+. □

LEMMA 22. *Let* q *be any iteration of Step* D2, *and let* w *be the computed witness against a vector* $v \prec_{\mathrm{I}} v_q$. *If one of* w *and* $w - v$ *is in* P^{t-1}, w *and* $w - v$ *satisfy Condition* (i) *or* (ii).

Proof. Assume that w is in P^{t-1}. Then w is in $X \subset X^+$ and $w - v$ is in X^-. By Lemma 21, $(w - v) + v_q$ is in X^+. Since $w, w - v \in X^-$ and $(w - v) + v_q \in X^+$, Condition (ii) is satisfied.

Assume that $w-v$ is in P^{t-1}. Then $w-v$ is in X and w is in X^+. By Lemma 21, $w-v_q$ is in X^-, which satisfies Condition (i). □

LEMMA 23. *Invariant q holds during iteration q for $1 \leq q \leq p$, and all the duels in Step* D2 *are legal.*

Proof. By Lemma 20, all witnesses computed before Step D2 satisfy Lemma 22. Thus the invariant holds at the beginning of iteration 1. Assume that it holds at the beginning of iteration $q \geq 1$. If Step D2 performs duels at iteration q, each duel is $\langle u : v_q \rangle$ for $u \in \hat{V}$. Consider a duel $\langle u : v_q \rangle$. We show that the duel $\langle u : v_q \rangle$ is legal and the new witness computed in the duel satisfies Invariant q:

Let $v = v_q - u$, and w be the witness against v. Note that $w+u$ $(w-v_q)$ is the apex of the type-1 (type-2) duel and $w-v = w-v_q+u$ is the cowitness against v (i.e. $w, w-v, w+u, w-v_q$ are the four points in the parallelogram of Fig. 10). The following pertain to the cases when witness w satisfies each of the conditions in Invariant q:

1. *Condition* (i). There are two cases.
   1.1. The type-1 duel is chosen by Rule 2. Since w and $w-v$ are in X^+ and w is closer to the backward diagonal $\hat{m}-1$ than $w-v$, $w-v$ is in X. Since $w-v \in X$ and $v_q \in C_{\mathrm{I}}^t$, the apex $w+u = (w-v)+v_q$ is in X^+ (the type-1 duel is legal). By Fact 2, $w+u$ is the new witness against u or v_q, and the new cowitness is $w-v$ (if v_q loses) or w (if u loses). In both cases, the new cowitness is in X^+. Since $(w+u)-v_q = w-v \in X \subset X^-$, Condition (i) is satisfied.
   1.2. The type-2 duel is chosen by Rule 2. The apex $w-v_q$ is in X^- by Invariant q (the type-2 duel is legal). By Fact 2, $w-v_q$ is the new cowitness, and the new witness $\hat{w}$ is w (if v_q loses) or $w-v$ (if u loses). If the new cowitness $w-v_q$ is in X, the new witness $\hat{w}$ and the new cowitness are in X^+, and $\hat{w}-v_q \in X^-$ because $(w-v)-v_q = (w-v_q+u)-v_q \prec_{\mathrm{I}} w-v_q \in X^-$ by $u \prec_{\mathrm{I}} v_q$. Hence Condition (i) is satisfied. If the new cowitness $w-v_q$ is in Z^-, we have the following two cases. If the new witness $\hat{w}$ is in X, then we have $(w-v_q)+v_q = w \in X^+$ and Condition (ii) is satisfied. Otherwise, the new witness is in Z^+ and Condition (iii) is satisfied.
2. *Condition* (ii). This is symmetric to the case of Condition (i).
3. *Condition* (iii). Since $w \in Z^+$ and $w-v \in Z^-$ and $v \prec_{\mathrm{I}} v_q \in C_{\mathrm{I}}^t$, $w+u = w-v+v_q$ $(w-v_q)$ is in Z^+ (Z^-) by Lemma 21. Both types of duels are legal. Without loss of generality, assume that the type-1 duel is chosen by Rule 2 (the other case is symmetric). By Fact 2, $w+u \in Z^+$ is the new witness. If v_q loses in the duel, Condition (iii) is satisfied since the new cowitness $w-v$ is in Z^-. If u loses, both the new witness $w+u$ and the new cowitness w are in $Z^+ \subset X^+$. Since $(w+u)-v_q = w-v \in Z^- \subset X^-$, Condition (i) is satisfied.

If w satisfies Condition (i), $w-v_{q'} \in X^-$ for $v_q \prec_{\mathrm{I}} v_{q'} \in C_{\mathrm{I}}^t$ by Lemma 21. Similarly, if w satisfies Condition (ii), $(w-v)+v_{q'} \in X^+$. Thus if a witness w satisfies Invariant q, w also satisfies Invariant q' for $q < q'$. Hence all witnesses against $v \prec_{\mathrm{I}} v_{q+1}$ which have been computed during iterations $1, \ldots, q$ satisfy Invariant $q+1$. Since all witnesses against $v \prec_{\mathrm{I}} v_{q+1}$ which were computed before Step D2 satisfy Invariant $q+1$ by Lemmas 20 and 22, all witnesses against $v \prec_{\mathrm{I}} v_{q+1}$ satisfy Invariant $q+1$. □

Step B3. We now consider the remaining case and assume that there are surviving candidates in C_{II}^t. Let $\hat{v}_{\mathrm{II}}$ be the shortest quad-II period of P^{t-1} (i.e., $\hat{m}/8 \leq |\hat{v}_{\mathrm{II}}| < \hat{m}/4$). By Lemma 16, all candidates in Q_{I}^1 and all the surviving candidates in C_{II}^t are lattice points on $v_{\mathrm{I}}, \hat{v}_{\mathrm{II}}$. We consider the quad-I vectors in C_{I}^t.

B3.1. Perform duels in each of Q_{I}^2, Q_{I}^3, and Q_{I}^4 using WITNESS(C_{I}^{t-1}) and WITNESS(C_{II}^{t-1}). The surviving candidates will form a line in each quadrant (at most four lines in C_{I}^t including the one in Q_{I}^1).

B3.2. For each line L in Q_{I}^2, Q_{I}^3, and Q_{I}^4, check if the candidates in L are lattice points on $v_{\mathrm{I}}, \hat{v}_{\mathrm{II}}$ by checking only one point in L because the points in L are lattice-congruent.

If all the candidates in C_{I}^t are lattice points, checking the lattice points of C_{I}^t and C_{II}^t is the same as Steps A2–A5 in the lattice-periodic case since P^{t-1} is lattice-generative with basis vectors $v_{\mathrm{I}}, \hat{v}_{\mathrm{II}}$. Notice that witnesses against nonlattice points have already been computed by duels.

Step B4. Assume that some candidates in C_{I}^t are nonlattice points on $v_{\mathrm{I}}, \hat{v}_{\mathrm{II}}$. Since all candidates in Q_{I}^1 are lattice points, nonlattice points are in Q_{I}^2, Q_{I}^3, and Q_{I}^4. If Q_{I}^2 and Q_{I}^4 contain nonlattice points, we can find witnesses against them as follows. Consider the line L in Q_{I}^2. By Theorem 1 with P^{t-1}, $v_{\mathrm{I}}, \hat{v}_{\mathrm{II}}$, and $z = (\lceil \hat{m}/4 \rceil - 1, \lceil \hat{m}/8 \rceil - 1)$, nonlattice points in L are not periods of P^{t-1}. We choose the longest vector v in L and find a witness w of P^{t-1} against v by symbol comparisons (i.e., $P^t[w] \neq P^t[w-v]$). Any point in L is a quad-I vector, and it is $v - iv_{\mathrm{I}}$ for some integer i. Hence we have $w - v \prec_{\mathrm{I}} w - v + iv_{\mathrm{I}} \prec_{\mathrm{I}} w$. Since w and $w - v$ is in P^{t-1}, $w - v + iv_{\mathrm{I}}$ is also in P^{t-1}. By the line-periodicity of P^{t-1}, $P^t[w-v] = P^t[w - v + iv_{\mathrm{I}}] \neq P^t[w]$, i.e., w is a witness against all vectors of L. A similar situation holds for the line in Q_{I}^4.

By the computation so far, all the surviving candidates in C_{I}^t except possibly the line of candidates in Q_{I}^3 are lattice points. Since Step B3.1 above is the same as Step D1.1, all the witnesses computed before and during Step B3.1 satisfy Lemma 19. Since we find only witnesses of P^{t-1} in the previous paragraph, all the witnesses computed so far (against vectors in C_{I}^t) satisfy Lemma 19.

Let $\hat{L}$ be the line of candidates in Q_{I}^3. Assume that the candidates in $\hat{L}$ are nonlattice points. Check if the longest vector v in $\hat{L}$ is a quad-I period of P^{t-1} by symbol comparisons. Notice that nonlattice point v can be a period of P^{t-1} because Theorem 1 does not apply to this case. If v is not a period, we can find witnesses against all points in $\hat{L}$ as we did for the line L in Q_{I}^2 above and go to the lattice-periodic case with the remaining candidates (lattice points) in C_{I}^t and C_{II}^t. Thus assume that nonlattice point v is a period of P^{t-1}. Then we have the following special properties.

LEMMA 24. (1) *The quad*-II *vector* $\hat{v}_{\mathrm{II}}$ *is in* Q_{II}^4 *and it is the only remaining candidate in* C_{II}^t.

(2) *There is no pair of candidates* u_1, u_2 *in* C_{I}^t *such that* $u_1 - u_2 = \hat{v}_{\mathrm{II}}$.

Proof. (1) $\hat{v}_{\mathrm{II}}$ is not in Q_{II}^1 because otherwise nonlattice points in C_{I}^t would not be periods of P^{t-1} by Theorem 1 with P^{t-1}, $v_{\mathrm{I}}, \hat{v}_{\mathrm{II}}$, and $z = (\lceil \hat{m}/4 \rceil - 1, \lceil \hat{m}/4 \rceil - 1)$. Similarly, $\hat{v}_{\mathrm{II}}$ is not in Q_{II}^3. Thus $\hat{v}_{\mathrm{II}}$ is in Q_{II}^4.

Suppose that there is a candidate $u \in C_{\mathrm{II}}^t$ which is not $\hat{v}_{\mathrm{II}}$. Since u is a lattice point by Lemma 16 and $\hat{v}_{\mathrm{II}} \in Q_{\mathrm{II}}^4$, u is $\hat{v}_{\mathrm{II}} + iv_{\mathrm{I}}$ for $i \neq 0$. If $i > 0$, the sum of the column coordinates of $\hat{v}_{\mathrm{II}}$ and v_{I} is less than $\hat{m}/4$, and the row coordinate of $\hat{v}_{\mathrm{II}} \in C_{\mathrm{II}}^t$ is larger than that of v_{I}, C_{II}^t (thus a quadrant of P^{t-1}) covers the unit cell on $v_{\mathrm{I}}, \hat{v}_{\mathrm{II}}$. Similarly, if $i < 0$, C_{II}^t covers the unit cell. By Theorem 1, nonlattice points in C_{I}^t cannot be periods of P^{t-1}, contradicting the existence of v.

(2) Suppose that there is a pair of candidates u_1, u_2 in C_{I}^t such that $u_1 - u_2 = \hat{v}_{\mathrm{II}}$. Since $\hat{v}_{\mathrm{II}}$ is in Q_{II}^4, u_1 is in Q_{I}^4 and u_2 is in Q_{I}^2. Since all candidates in Q_{I}^2 and Q_{I}^4 are lattice points, so are u_1 and u_2. Thus one of u_1 and u_2 is iv_{I} and the other is $iv_{\mathrm{I}} \pm \hat{v}_{\mathrm{II}}$

for $i > 0$, which implies (as in (1)) that C_{I}^t covers the unit cell on $v_{\mathrm{I}}, \hat{v}_{\mathrm{II}}$. By Theorem 1, we get a contradiction to the existence of v. □

To find the valid quad-I periods of P^t, run Step D1.2 and then Step D2 and procedure LINE with the candidates in C_{I}^t. Note that all the witnesses against points in C_{I}^t (C_{II}^t) satisfy Lemma 19 (Lemma 17). During Step D1.2, all the duels have witnesses by Lemma 24. To find the valid quad-II periods of P^t, check if $\hat{v}_{\mathrm{II}}$, which is the only candidate, is a valid period of P^t by symbol comparisons.

4.4. P^{t-1} is radiant-periodic (quad-I).

If P^{t-1} is radiant-periodic in quad-I, P^{t-1} has no quad-II period v such that $|v| < \hat{m}/4$ by Lemma 9, which implies P^t has no valid quad-II periods; P^t is not lattice-periodic. P^t is not radiant-periodic in quad-I because otherwise P^{t-1} (which is contained in $F_{\mathrm{I}}(P^t)$) would be lattice-generative with basis vectors b_1 and b_2 satisfying $|b_1|, |b_2| < \hat{m}/4$ by Lemmas 7 and 11, contradicting the fact that P^{t-1} has no quad-II period v such that $|v| < \hat{m}/4$. Thus P^t is either line-periodic in quad-I or nonperiodic.

The treatment of the radiant-periodic case is almost an exact analogue to that of the line-periodic case, replacing "lines" by "monotone lines" in all the places except the line of candidates produced by Step D2 (which is the input to procedure LINE). The only difference is that, as noted above, P^{t-1} has no quad-II periods of length $< \hat{m}/4$, and we only have Steps B1 and B2.

THEOREM 4. *WITNESS(C_{I}) and WITNESS(C_{II}) of the array P can be computed in $O(m^2)$ time.*

Proof. The correctness follows from the discussion above. Since stage t takes $O(\hat{m}^2)$ time, the overall time is $O(m^2)$. □

5. Conclusion.

The problem of designing an alphabet-independent two-dimensional pattern-matching algorithm has been open for quite some time. It was partially solved in [4], and in this paper, we completely solve it by designing an alphabet-independent linear-time algorithm for two-dimensional witness computation. There are several other string algorithms (notably those that use suffix trees) which are alphabet-dependent, and it will be nice to either find an alphabet-independent algorithm or prove that the dependence on the alphabet is inherent.

Acknowledgments. We thank Amihood Amir for helpful suggestions and the anonymous referees for numerous comments.

REFERENCES

[1] A. V. AHO AND M. J. CORASICK, *Efficient string matching: An aid to bibliographic search*, Comm. Assoc. Comput. Mach., 18 (1975), pp. 333–340.

[2] A. AMIR AND G. BENSON, *Two-dimensional periodicity in rectangular arrays*, manuscript, 1991.

[3] ———, *Two-dimensional periodicity and its applications*, in Proc. 3rd ACM–SIAM Symposium on Discrete Algorithms, Society for Industrial and Applied Mathematics, Philadelphia, 1992, pp. 440–452.

[4] A. AMIR, G. BENSON, AND M. FARACH, *An alphabet independent approach to two-dimensional pattern matching*, SIAM J. Comput., 23 (1994), pp. 313–323.

[5] A. AMIR, G. LANDAU, AND U. VISHKIN, *Efficient pattern matching with scaling*, J. Algorithms, 13 (1992), pp. 2–32.

[6] T. J. BAKER, *A technique for extending rapid exact-match string matching to arrays of more than one dimension*, SIAM J. Comput., 7 (1978), pp. 533–541.

[7] R. S. BIRD, *Two dimensional pattern matching*, Inform. Process. Lett., 6 (1977), pp. 168–170.

[8] D. Breslauer and Z. Galil, *An optimal $O(\log\log n)$ time parallel string matching algorithm*, SIAM J. Comput., 19 (1990), pp. 1051–1058.

[9] R. M. Karp, R. E. Miller, and A. L. Rosenberg, *Rapid identification of repeated patterns in string, trees, and arrays*, in Proc. 4th ACM Symposium Theory of Computing, Association for Computing Machinery, New York, 1972, pp. 125–136.

[10] R. M. Karp and M. O. Rabin, *Efficient randomized pattern-matching algorithms*, IBM J. Res. Develop., (1987), pp. 249–260.

[11] D. E. Knuth, J. H. Morris, and V. B. Pratt, *Fast pattern matching in strings*, SIAM J. Comput., 6 (1977), pp. 323–350.

[12] M. G. Main and R. J. Lorentz, *An $O(n \log n)$ algorithm for finding all repetitions in a string*, J. Algorithms, 5 (1984), pp. 422–432.

[13] U. Vishkin, *Optimal parallel pattern matching in strings*, Inform. and Control, 67 (1985), pp. 91–113.

[14] R. F. Zhu and T. Takaoka, *A technique for two-dimensional pattern matching*, Comm. Assoc. Comput. Mach., 32 (1989), pp. 1110–1120.

SIAM J. COMPUT.
Vol. 25, No. 5, pp. 936–955, October 1996

THE TREE MODEL FOR HASHING: LOWER AND UPPER BOUNDS*

JOSEPH GIL†, FRIEDHELM MEYER AUF DER HEIDE‡, AND AVI WIGDERSON§

Abstract. We define a new simple and general model for hashing. The basic model together with several variants capture many natural (sequential and parallel) hashing algorithms and represent common hashing practice. Our main results exhibit tight tradeoffs between hash-table size and the number of applications of a hash function on a single key.

Key words. parallel algorithms, randomization data structures

AMS subject classifications. 68P10, 68P05, 68Q10, 68Q22

1. Introduction.

Hashing is one of the most important concepts in computer science. Its applications touch almost every aspect of this field—operating systems, file-structure organization [17], communication, parallel and distributed computation, efficient algorithm design, and even complexity theory [26, 27]. Nevertheless, the most common use of hashing is for the very fundamental question of efficient storage of sparse tables (see [23] and [19] for a systematic study).

A fundamental result of Fredman, Komlós, and Szemerédi [10] shows that n elements (keys) from a universe of any size can be hashed to a linear-size table in linear expected time, allowing for constant search time. It was observed [21], however, that although the average insertion time per element is constant, parallel application of this algorithm does not work in constant time. The reason is that while the average is constant, some elements will have to be hashed a nonconstant number of times.

In this paper, we study the question of whether parallel hashing can be done in constant time. We present a simple new general model that captures many natural (sequential and parallel) hashing algorithms. In a game against nature, the algorithm and coin tosses cause the evolution of a random tree whose size corresponds to space (hash-table size) and two notions of depth correspond to the longest probe sequences for insertion (parallel insertion time) and search of a key, respectively.

We study these parameters of hashing schemes by analyzing the underlying stochastic process and derive tight lower and upper bounds on the relation between the amount of memory allocated to the hashing execution and the worst-case insertion time. In particular, we show that except for extremely unlikely events, every input set of size n will have members for which $\Omega(\lg\lg n)$ applications of a hash function are required. From a parallel perspective, we obtain that if n processors are each given a key drawn from a large universe and if the input keys cannot be exchanged among the processors, then $\Omega(\lg\lg n)$ expected time is required to hash the input keys into $O(n)$ space. This is despite the existence of serial algorithms which achieve constant amortized time for insertion as well as constant worst-case search time [8].

* Received by the editors September 17, 1993; accepted for publication (in revised form) January 13, 1995. This work was included in the first author's Ph.D. dissertation [11] and appeared in a preliminary form in [14]. This research was supported in part by the Leibniz Center for Research in Computer Science.

† Computer Science Department, Technion–Israel Institute of Technology, Technion City, Haifa 32000, Israel (yogi@cs.technion.ac.il).

‡ Heinz Nixdorf Institut, Fachbereich Mathematik/Informatik, Universität Paderborn, D-33095 Paderborn, Germany (fmadh@uni-paderborn.de).

§ Computer Science Department, Hebrew University of Jerusalem, Jerusalem 91904, Israel (avi@cs.huji.ac.il).

Three variants of the basic model, which represent common hashing practice, are defined, and tight bounds are presented for them as well. The most striking conclusion that can be drawn from the bounds is that, under all combinations of model variants, not all keys may be hashed in constant time.

Outline. This paper is organized as follows. Section 2 describes the basic model and its variants. Section 3 presents our lower- and upper-bound results. These bounds are proved in §§4 (lower bounds) and 5 (upper bounds). Finally, concluding remarks are given in §6.

2. The tree model for hashing.

2.1. The basic model.

The process of inserting a set S of n elements taken from some universe U into a hash table can be thought of as a process of refining partitions and is depicted simply by a tree. Originally, all elements reside in a single node (the root). A hashing algorithm begins by choosing a range size m and then selecting a *hash function* h which maps U to the range $[m]$. Selection of h is according to some distribution defined on $U^{[m]}$. It partitions S into subsets $S_1, S_2, \ldots, S_m$ (some empty) such that $S_i = h^{-1}(i) \cap S$, $i = 1, \ldots, m$. Following the literature, these subsets are sometimes called *buckets*. The function h is stored at the root of the tree. The root has m children; the subset S_i is moved to the ith child of the root.

The process is repeated for each S_i that contains at least two elements. The refining halts when every leaf contains at most one element from S.

A search for an element $x \in U$ in the hash table is easily performed by following the path from the root which is determined by applying the hash functions at internal nodes to the element x and, when reaching a leaf, comparing x to the element residing there if such an element exists.

This model leads to an alternate view of a hashing algorithm as an element-distinctness-proof generator. The input is a set of distinct keys taken from a universe with no order relation defined on it. The output is a proof that all elements are distinct. The proof components are functions from the universe to a bounded range.

2.2. Comments.

1. The two types of strategic decisions made by a specific algorithm are the choice of range (i.e., number of children) for the hash function, and the choice of distribution used in selecting a particular function to this range. We *assume* that the function used is a truly random function, i.e., all possible functions are given equal probability.

This assumption follows the tradition in the design and analysis of hashing algorithms. Designers of hashing algorithms view truly random functions as an ideal which cannot be realized because of the huge space required for their representation. In almost all such analyses (see, e.g., [19, pp. 514–517] and [23, pp. 120–124] for textbook examples as well as [18, 29, 4, 5, 6, 24, 25]), it is assumed that the hash functions used are random or, alternately, that the functions are fixed, but the input set is selected at random. The reason for this assumption is that random functions have been intuitively perceived as the best for hashing [20]. This intuition was proved correct under quite general conditions by Ajtai, Komlós, and Szemerédi [1]. (It should be noted, however, that random functions are not always the best. For example, if $|U| = O(n)$, then the identity function might produce much fewer collisions than a truly random function.)

With our random-functions assumption, the hashing process that occurs in a node can be described as the act of independently sending each element of U to each possible child with uniform distribution. Analyzing the full hashing algorithm is reduced to analyzing a natural process of successively throwing identical balls into boxes until all the balls reside in distinct boxes.

2. Most common algorithms are stronger than the process we described—they use retries when the chosen hash function is extremely bad (e.g., all elements were mapped to one cell) and allow the storage of elements in the internal nodes of the tree as well as in the leaves. These generalizations and others will be considered later by introducing variants to the basic model.

3. We deliberately deal here with the static case, i.e., when all the elements to be inserted are known in advance. This does not restrict the generality of the lower bounds. However, the algorithms presented are for the static case only.

4. Yao's cell-probe model [30], the standard general model for hashing, can also be described as a tree in a similar way. Our model differs from his in the way that a decision tree differs from a Turing machine. The cell-probe model allows each cell a limited number of bits (depending on U), but these can encode arbitrary objects and be computed at no cost. Our cells contain either elements or functions. Functions can only be applied to elements and two elements can only be tested for equality. Our model, being more structured, is cleaner and easier to analyze, though less general.

2.3. Resources. We have seen that the stochastic process determined by a hashing algorithm Alg given $S \subset U$ of size n is described by a random tree. The main resources of Alg operating on S are natural parameters of this tree.

Space. The space required, or hash-table size, denoted by $\textsc{Space}(\mathsf{Alg}, S)$, is simply the total number of nodes in the tree. Note that the space resource also includes nodes with empty sets. This is in accordance with the standard way of measuring space complexity in hashing algorithms that charges for unused cells in the hash table.

Insertion time. We denote by $\textsc{Time}(\mathsf{Alg}, S)$ the total insertion time. This is the sum of depths of all leaves containing an element, i.e., the number of hash-function applications to all the elements. Each application counts as one time unit. (Indeed, many practical algorithms use hash functions which can be evaluated in constant time.)

Parallel insertion time. We denote the depth of the tree by $\textsc{Depth}(\mathsf{Alg}, S)$. This is the parallel insertion time under the assumption that each processor is assigned one key and this processor alone is responsible for inserting this key. This parameter has two important meanings for sequential algorithms as well. It captures the number of functions needed to resolve the "worst" pair of elements and the worst-case insert and search time.

Search time will not be identical to insertion time in the more general models, so we devote a different notation to it.

Maximum search time. This is the largest number of function applications needed to find out if $x \in U$ is in S using the tree generated by Alg on S. This parameter will be denoted by $\textsc{Search}(\mathsf{Alg}, S)$. In the basic model, this definition coincides with that of $\textsc{Depth}(\mathsf{Alg}, S)$. However, this will no longer be true when the model is elaborated, although we will still have $\textsc{Search}(\mathsf{Alg}, S) \leq \textsc{Depth}(\mathsf{Alg}, S)$.

Let $\textsc{Param}$ be a generic parameter ($\textsc{Space}$, $\textsc{Time}$, $\textsc{Depth}$, or $\textsc{Search}$); then $\overline{\textsc{Param}}(\mathsf{Alg}, S)$ will denote the expectation of $\textsc{Param}$ with respect to the random choices made by the algorithm Alg, so $\overline{\textsc{Param}}(\mathsf{Alg}, S) = \mathbf{E}(\textsc{Param}(\mathsf{Alg}, S))$. We denote by

$$\overline{\textsc{Param}}(\mathsf{Alg}, n) = \max_{|S|=n} \overline{\textsc{Param}}(\mathsf{Alg}, S)$$

the performance of Alg on a worst-case set S of size n and by

$$\overline{\text{PARAM}}(n) = \min_{\mathsf{Alg}} \overline{\text{PARAM}}(\mathsf{Alg}, n)$$

the performance of the best algorithm on its worst-case set S. At times, it will be useful to ignore the probabilistic performance and consider the worst possible performance of Alg (over all possible runs) on the worst-case input, which we denote by $\text{PARAM}(\mathsf{Alg}, n)$.

2.4. Variants of the basic model. Finally, we consider more powerful algorithms than those permitted by the basic model.

Retries. An algorithm may allocate (say) m boxes (children) for n balls residing at a node v and find that in throwing them randomly, they all fall into one or very few cells. This is an unlikely event that causes a waste of space. The algorithm is allowed to consider this (or other more likely events) "bad" and try again. We do not charge for space used in v. To maintain the meaning of depth in this variant, we create one single child for v and move all the balls there.

We attach the subscript r to the resource measures in this model, e.g., $\text{SPACE}_r(\mathsf{Alg}, S)$ and $\overline{\text{DEPTH}_r}(n)$, etc. Note that SEARCH_r may be much smaller than DEPTH_r since while a search is being performed, no function application should be done at a node with only one child.

Chaining. This variant allows the algorithm to store elements in internal nodes as well. Specifically, when m keys reach a node v, one of them is stored in v and the remaining $m - 1$ proceed to v's children. The term *chaining* is used since this variant generalizes hashing techniques in which a chain (a linked list) of keys can be stored in hashing-array positions.

The subscript c is added to the resources measures. Clearly SEARCH_c and DEPTH_c are the same again since even if no branching occurs at node v, the element being searched for should be compared to the one residing at v.

We allow a combination of chaining and retries which is denoted by the double subscript cr. In this combination, the algorithm may leave a key in an internal node even if the function used at this node was discarded. Obviously, such a node cannot be skipped in a search.

Parallel hashing. In this variant of the model, we allow the algorithm to try in parallel several hash functions in a node v and then pick one of them to create v's children. Space here is counted as the sum of ranges of *all* those functions. The subscript p is added to the resource measures in this variant.

This variant may be combined with the two previous ones; if retries are permitted, then the algorithm may choose not to use any of the hash functions that were tried; if exploiting internal nodes is possible, then the algorithm can leave one element at v no matter which hash function is selected for the node. Despite its name, this variant does not lead directly to a parallel random-access machine (PRAM) algorithm. The major difficulty is the assignment of processors that have completed handling their original key to assist the other processors with the yet unhashed keys.

One possible hashing variant was deliberately omitted from the above list; we do not permit the merging of nodes in the hash trees. Intuitively, merges *lose* separation information, and omitting merges from a hashing algorithm should only improve its performance. It is easy to verify that the TIME, DEPTH, and SEARCH complexity measures can only decrease as a result of eliminating merge operations. The only possible merit of merging is to SPACE. It will be evident from the lower-bound proofs that merging cannot improve an algorithm with respect to all of the complexity measures defined above.[1]

Most hashing algorithms deviate from our basic model by allowing one or both of the retries or the chaining variants. The parallel variant is mentioned as an alternate hashing idea in [12] and is used by Matias and Vishkin [22].

3. Results. The most interesting algorithms are those that achieve $\textsc{Space}(\mathsf{Alg}, n) = O(n)$, i.e., linear space. In his seminal paper [30], Yao asked if one can simultaneously achieve $\textsc{Space}(n) = O(n)$ and $\overline{\textsc{Search}}(n) = O(1)$. In our *basic* model, this is impossible.

THEOREM 3.1. *If* $\textsc{Space}(\mathsf{Alg}, n) = O(n)$, *then* $\overline{\textsc{Depth}}(\mathsf{Alg}, n) = \overline{\textsc{Search}}(\mathsf{Alg}, n) = \Omega(\lg\lg n)$.

(The proof for this theorem, as well as for all other results presented in this section is in subsequent sections.)

However, allowing retries, Yao gave an algorithm Y which achieves $\textsc{Space}_r(\mathsf{Y}, n) = O(n)$ and $\textsc{Search}_r(\mathsf{Y}, n) = O(1)$ for large enough universes. For small universes, $q = n^{O(1)}$, Tarjan and Yao [28] showed how linear storage and constant search time can be maintained. Fredman, Komlós, and Szemerédi [10] closed the gap by an algorithm FKS that satisfies $\textsc{Space}_r(\mathsf{FKS}, n) = O(n)$ and $\textsc{Search}_r(\mathsf{FKS}, n) = O(1)$ for *any* universe size and any input set. Analyzing their algorithm, we find that while insertion time $\overline{\textsc{Time}_r}(\mathsf{FKS}, n) = O(n)$, (i.e., on the average we apply only a constant number of functions to each element), $\overline{\textsc{Depth}_r}(\mathsf{FKS}, n) = \Omega(\lg n)$, so some element will be hashed $\Omega(\lg n)$ times, and this is the time required for hashing the elements in parallel using the FKS scheme.[2] A natural question that arises is whether this parameter can decrease to $O(1)$. We answer this question in the following theorem.

THEOREM 3.2. *If* $\textsc{Space}(\mathsf{Alg}, n) = O(n)$, *then* $\overline{\textsc{Depth}_r}(\mathsf{Alg}, n) = \Omega(\lg\lg n)$.

Remark. There exists an algorithm DM, due to Dietzfelbinger and Meyer auf der Heide [9], for managing a dynamic-data hash table that achieves with very high probability constant worse-case performance. However, DM does not contradict the stated lower bounds since it fits into neither our basic model nor any of its variants. In particular, DM pipelines the insertions; processing an inserted element can continue for up to n^ϵ steps after the insertion takes place; the algorithm allows keys to be fetched even if they are not "fully" inserted. Still, as the lower bounds indicate, there is no easy way of constructing a fast parallel version of DM. There are always keys for which DM requires as many as n^ϵ function applications.

With the help of retries, the lower bound of Theorem 3.1 can be met.

THEOREM 3.3. *There is an algorithm* RetryShallow *which uses linear space* (*i.e.*, $\textsc{Space}_r(\mathsf{RetryShallow}, n) = O(n)$) *and gives*

(i) $\overline{\textsc{Time}_r}(\mathsf{RetryShallow}, n) = O(n)$,

(ii) $\overline{\textsc{Depth}_r}(\mathsf{RetryShallow}, n) = O(\lg\lg n)$, *and*

(iii) $\textsc{Search}_r(\mathsf{RetryShallow}, n) = O(1)$.

This algorithm is a variant of the FKS algorithm. The improvement in $\overline{\textsc{Depth}}(n)$ is accomplished by using a different, more adaptive memory-allocation scheme while executing the retries. This algorithm is optimal with respect to all parameters even if we count arithmetic operations and limit word size to $O(\lg|U|)$. Moreover, if we

[1] However, it is interesting to note that the merging technique is useful for the construction of good pseudorandom functions which may be used for implementing hashing algorithms [9].

[2] Indeed, the parallel hashing scheme of Matias and Vishkin [22], being based directly on FKS, takes $O(\lg n)$ parallel time.

restrict the algorithm to the basic model by eliminating retries, then all the parameters (except for $\textsc{Search}(\mathsf{Alg})$, which will be the same as $\textsc{Depth}(\mathsf{Alg})$) will remain optimal.

THEOREM 3.4. *There is an algorithm* BasicShallow *for which*

(i) $\overline{\textsc{Space}}(\mathsf{BasicShallow}, n) = O(n)$,

(ii) $\overline{\textsc{Time}}(\mathsf{BasicShallow}, n) = O(n)$, *and*

(iii) $\overline{\textsc{Depth}}(\mathsf{BasicShallow}, n) = \overline{\textsc{Search}}(\mathsf{BasicShallow}, n) = O(\lg\lg n)$.

A nontrivial worst-case upper bound for SPACE is not possible here because for any such bound, there are (admittedly rare) cases in which enough failures occur to force an algorithm to overflow this bound.

The general tradeoff between space and depth is given by the following theorem.

THEOREM 3.5. *If* $\textsc{Space}(\mathsf{Alg}, n) = n^{1+1/\lambda}$, *then* $\overline{\textsc{Depth}_r}(\mathsf{Alg}, n) = \Omega(\lg\lambda)$.

Can the common practice of using internal nodes for storage help by more than a constant factor? Again, perhaps surprisingly, the answer is positive.

THEOREM 3.6. *Both* $\textsc{Space}_c(n) = O(n)$ *and* $\overline{\textsc{Depth}_c}(n) = O(\lg\lg n/\lg\lg\lg n)$ *can be achieved simultaneously.*

The algorithm behind this theorem uses truly random hash functions or, equivalently, high-degree polynomials. As a more practical alternative, the class of pseudorandom hash functions defined in [9] can be used here as well.

The next theorem shows that this meager improvement of the $\lg\lg\lg n$ factor is the best possible, and even it cannot coexist with the employment of retries to achieve $O(1)$ search time. (As before, adding the power of retries to this variant of the model cannot improve $\overline{\textsc{Depth}}(n)$.)

THEOREM 3.7. *Let* Alg' *be a hashing algorithm operating in the chaining-model retries and let* Alg *be the same algorithm restricted to the chaining model. Then we have the following:*

(a) *If* $\textsc{Space}_{cr}(\mathsf{Alg}', n) = \textsc{Space}_c(\mathsf{Alg}, n)$, *then*

$$\overline{\textsc{Depth}_{cr}}(\mathsf{Alg}', n) = \Omega\big(\overline{\textsc{Depth}_c}(\mathsf{Alg}, n)\big).$$

(b) *If* $\textsc{Space}_c(\mathsf{Alg}, n) = O(n)$, *then*

$$\Omega\big(\overline{\textsc{Depth}_c}(\mathsf{Alg}, n)\big) = \Omega\big(\overline{\textsc{Search}_c}(\mathsf{Alg}, n)\big) = \Omega(\lg\lg n/\lg\lg\lg n).$$

(c) *If* $\textsc{Space}_{cr}(\mathsf{Alg}', n) = \textsc{Space}_c(\mathsf{Alg}, n) = O(n)$ *and* $\overline{\textsc{Depth}_{cr}}(\mathsf{Alg}, n) = O(\lg\lg n/\lg\lg\lg n)$, *then*

$$\overline{\textsc{Search}_{cr}}(\mathsf{Alg}', n) = \Omega\big(\overline{\textsc{Depth}_c}(\mathsf{Alg}, n)\big).$$

The general tradeoff is given by the following theorem.

THEOREM 3.8. *If* $\textsc{Space}_c(\mathsf{Alg}, n) = n^{1+1/\lambda}$, *then* $\overline{\textsc{Depth}_c}(\mathsf{Alg}, n) = \Omega(\lg\lambda/\lg\lg\lambda)$.

In a clear contrast to the first two variants, the "simultaneous retries," which may be applied in the parallel variant, lead to a significant improvement in DEPTH because they allow the folding of many iterations into one. Nevertheless, constant time hashing cannot be achieved in this case as well.

THEOREM 3.9. *If* $\textsc{Space}_p(\mathsf{Alg}, n) = O(n)$, *then* $\overline{\textsc{Depth}_p}(\mathsf{Alg}, n) = \Theta(\lg^* n)$.

Neither retries nor chaining can further decrease the maximal insertion time of the parallel variant.

THEOREM 3.10. *If memory usage is restricted to* $O(n)$, *then*

$$\overline{\textsc{Depth}_{rcp}}(n) = \Omega\big(\overline{\textsc{Depth}_p}(n)\big).$$

4. Proofs of lower bounds. We view hashing algorithms in the treee model from a parallel perspective. Each parallel iteration is an attempt to separate *all* subsets of S that were not previously separated, i.e., subsets that still have two or more keys in them. Thus successive iterations correspond to successive tree levels.

It should be obvious that with the usage of truly random functions, the performance of the algorithm is dependent on the size of S but not on its content. Let Opt be the best possible algorithm for the current setting of the parameters (space and model variant). Our proofs are based upon showing that, with a dominant probability, there is a minimal number of iterations that Opt has to go through.

For simplicity in the analysis, we let Opt make the following assumptions.

Extra memory. Say that the problem restricts the memory usage to a total of m memory cells. This restriction will be weakened for Opt and it will be allowed to use m memory cells in *each* iteration.

Partial separation. A mapping of a set of keys to memory is called a *partial separation* if there exist two keys in the set that are mapped to distinct cells. Opt may consider any partial separation as being a total separation. The extremely unlikely case in which all keys from the set are mapped to the same memory is called a *complete failure*. Only complete failures need to be passed to the next iteration of Opt.

Restricted set size. In iteration t, Opt has only to deal with (nodes containing) sets of r_t keys. Smaller or larger sets can be completely ignored. The exact value of r_t will be specified later.

Early termination. Opt need not be concerned with the case where there are fewer than $\lg n$ sets of size r_t. As soon as the number of sets drops below that bound, Opt can terminate immediately.

Higher success probability. While analyzing Opt, we will assume that failure probability is determined by $r = r_1$, although r_t keys are actually mapped. It will be shown that this assumption may only decrease the failure probability and works in Opt favor.

To account for the random nature of the hashing process, the following definition is introduced.

DEFINITION 4.1. *Events that occur with probability smaller than $n^{-\epsilon}$ for some $\epsilon > 0$ are called* negligible *events.* Dominating *events are the complement of negligible events.*

Negligible events will be ignored in the following discussion since even if they could be treated by Opt without *any* resource investment, the expected value of the performance measures will essentially be the same.

The rest of this section is outlined as follows. Suitable values for r_t will be set. Then we will compute a lower bound on the initial number of sets of size r_1. (Since the algorithm is based on a random process, it may be extremely lucky and break this bound; thus the lower-bound statement should be read with "ignoring negligible events" appended to it. Such quantification is implied henceforth.) We next estimate the number of sets of size r_{t+1} in iteration $t+1$ as a function of the number of sets of size r_t in iteration t. Then an *explicit* lower-bound for the number of sets of size r_t in iteration t is derived. The lower-bound proofs are then completed by computing the minimal number of iterations Opt must undergo before completion. The analysis is done for the basic model and the chaining variant together and then it is repeated for the parallel variant. We conclude with a remark explaining why all lower bound proofs are applicable to all the retries variants.

4.1. Root-node hashing. The root node corresponds to the $0th$ iteration. In it the set S of n keys is separated into m subsets using a random function $h : U \mapsto [m]$

into subsets $S_1, S_2, \ldots, S_m$. Since h is a random function, the root node is accurately modeled by the well-studied "balls-into-urns" model [16].

Let $\alpha = n/m$. For our needs, it is sufficient to restrict attention to the case $\alpha = O(1)$. It is easy to verify that as n tends to infinity, the distribution of the number of balls (keys) in any single urn (cell) approaches the Poisson distribution parametrized by α. Let $N(r)$ be the number of subsets that have exactly r elements in them. Then there exists n' such that for every $n > n'$,

$$\mathbf{E}\left(N(r)\right) > \frac{1}{2} m e^{-\alpha} \frac{\alpha^r}{r!}. \tag{1}$$

Without loss of generality, assume that $n > n'$ from now on.

To see that $N(r)$ is "tightly concentrated" around its expectation, we need the following fact.

FACT 1 (Azuma). *Let $F(x_1, \ldots, x_n)$ be an arbitrary function of n variables which satisfies*

$$|F(x_1, \ldots, x_n) - F(x_1, \ldots, x_{i-1}, x'_i, x_{i+1}, \ldots, x_n)| \leq 1$$

for any setting of i, x'_i, and $x_1, \ldots, x_n$. Then if $x_1, \ldots, x_n$ are independent random variables,

$$\text{Prob}\left(|F - \mathbf{E}(F)| > \lambda\sqrt{n}\right) < e^{-\lambda^2/2}.$$

Proof. A proof for Azuma's inequality can be found in [2, Chap. 7]. This textbook also presents its usage as a general technique in random graphs. □

For $i = 1, \ldots, n$, let x_i be the the cell into which ith member of S was mapped. Consider the function $N(r)/2$ to be dependent on the x_i's. A change in x_i can change the value of $N(r)/2$. Therefore, we can apply Fact 1 to obtain that the probability of $N(r) < \mathbf{E}(N(r))/2$ is negligible. Consequently,

$$N(r) \geq \frac{1}{4} m e^{-\alpha} \frac{\alpha^r}{r!}, \tag{2}$$

except for a negligible number of cases.

4.2. The basic model and the chaining variant. In the basic model, we follow only sets of size 2, i.e., $r_t = 2$ for $t \geq 1$. When the usage of intermediate nodes (chaining) is possible, sets of fixed size r can no longer be tracked since the number of iterations will depend on n, and even complete failure to hash a set will decrease its size by 1. Instead, define $r_0 = r = r(n)$ and $r_{t+1} = r_t - 1$. Note that in this variant, r_0 must be greater than the desired lower bound for the number of iterations.

The following fact estimates $\mathbf{E}(N(r))$ for those pairs of r and m in which we are interested. Note that in all of the cases below, $\mathbf{E}(N(r))$ is $\Omega(n^\epsilon)$ for some $\epsilon > 0$ and hence the event $N(r) < \mathbf{E}(N(r))/2$ is negligible.

LEMMA 4.2. *The expected value of $N_0(r)$, the initial (after the root-node hashing) number of sets of size r, is given by the following:*

1. *If $r = 2$ and $m = O(n)$, then*

$$\mathbf{E}\big(N_0(r)\big) = \Omega(n).$$

2. *If* $r = 2$ *and* $m = n^{1+1/\lambda}$ *for some fixed* λ, *then*

$$\mathbf{E}\big(N_0(r)\big) = \Omega\Big(n^{1-1/\lambda-1/n^{1/\lambda}\ln n}\Big) = \Omega\Big(n^{1-1/\lambda-o(1)}\Big).$$

3. *If* $r = \lg\lg n/\lg\lg\lg n$ *and* $m = O(n)$, *then*

$$\mathbf{E}\big(N_0(r)\big) = \Omega\Big(n^{1-r(\lg r-\lg\alpha)/\lg n)}\Big) = \Omega\Big(n^{1-o(1/\lg n)}\Big).$$

4. *If* $r = \lg\lambda/\lg\lg\lambda$ *and* $m = n^{1+1/\lambda}$ *for some fixed* λ, *then*

$$\mathbf{E}\big(N_0(r)\big) = \Omega\Big(n^{1-(r-1)/\lambda-o(1)}\Big).$$

Proof. Apply inequality (1). □

From the simplifying assumptions it follows that in iteration t, Opt uses m memory cells to deal with N_t sets of r_t keys each. The algorithm should allocate memory to cells in a way that will minimize the number of failures N_{t+1}. The following lemma reveals the memory-allocation scheme used by Opt.

LEMMA 4.3. Opt *uses a balanced memory-allocation scheme*; *each of the* N_t *sets is hashed into* m/N_t *cells.*

Proof. In iteration t, if a subset (that has r_t keys) is mapped by a random function into m_i memory cells, then the *complete-failure* probability is $m_i^{1-r_t}$. This probability is a decreasing function of m_i; therefore, a memory allocation does not minimize the failure probability unless all the m cells are utilized. Let $m = m_1 + m_2 + \cdots + m_{N_t}$ be a memory allocation of the m cells to the N_t sets. The expected value of N_{t+1} is given by

$$\mathbf{E}\left(N_{t+1}\right) = \sum_{i=1}^{N_t} m_i^{1-r_t},$$

and by convexity, this is minimized when all m_i's are equal. □

The complete-failure probability $m_i^{1-r_t}$ increases as r_t decrease. Thus it is permissible to assume that Opt uses a complete-failure probability derived from $r = r_1$, the initial size of the sets. We can then write

$$\mathbf{E}\left(N_{t+1}\right) = \sum_{i=1}^{N_t} m_i^{1-r},$$

and by Lemma 4.3,

$$\mathbf{E}\left(N_{t+1}\right) = N_t\left(\frac{m}{N_t}\right)^{1-r}.$$

The probability that N_{t+1} will be much smaller than its expected value is estimated by the following lemma.

LEMMA 4.4. *Let* N_t *be fixed. The event* $N_{t+1} < \mathbf{E}\left(N_{t+1}\right)/4$ *is n-negligible if* $\mathbf{E}\left(N_{t+1}\right) > \lg n$.

Proof. Note that N_{t+1} is the sum of N_t *independent* random binary variables. The lemma is obtained from application of Chernoff bounds [7]. (See [3, 15] for a succinct statement of these bounds.) □

Thus we can assume that $N_t \geq \mathbf{E}(N_t)/4$ simultaneously in all iterations. For simplicity, we permit Opt to have

$$N_{t+1} = \frac{N_t}{4}\left(\frac{m}{N_t}\right)^{1-r}.$$

Let $\eta_t = N_t/m$. Then by dividing the above by m, we have

$$\eta_{t+1} = \frac{\eta_t^r}{4}.$$

This representation demonstrates the fact that the fraction of sets of a given size decreases "only" double-exponentially, giving rise to the double-logarithmic lower and upper bounds. The exact solution of the above recurrence is given by

$$\eta_t = \frac{\eta_0^{r^t}}{4^{(r^t-1)/(r-1)}}.$$

Overestimating η_t may only weaken the lower bound. We can therefore do so by writing

$$\eta_t \leq \left(\frac{\eta_0}{4}\right)^{r^t},$$

which facilitates an easy counting of the minimal number of iterations.

LEMMA 4.5. *If $m \leq n^3$, then the number of levels in Opt's tree is*

$$\Omega\left(\frac{1}{\lg r}\lg\frac{\lg m}{2-\lg\eta_0}\right).$$

Proof. Let T be given by

$$T = \frac{1}{\lg r}\lg\frac{\lg\lg n - \lg m}{\lg\eta_0 - 2}.$$

Then for $t < T$,

$$N_t = m\eta_t = m\left(\frac{\eta_0}{4}\right)^{r^t} > m\left(\frac{\eta_0}{4}\right)^{r^T} = \lg n.$$

It follows that if Opt executes less than T iterations, it will have more than $\lg n$ sets and it cannot terminate. The proof is completed by noting that for $m \leq n^3$,

$$T = \Omega\left(\frac{1}{\lg r}\lg\frac{\lg m}{2-\lg\eta_0}\right). \qquad \square$$

Applying this lemma to the estimates in Lemma 4.2 will yield the proofs for the lower bounds set by Theorems 3.1, 3.5, 3.7(b), and 3.8. In particular, we have the following:

- *Theorem* 3.1. Setting $r = 2$ and $m = O(n)$, we have $-\lg\eta_0 = O(1)$ and hence

$$\begin{aligned}\overline{\text{DEPTH}}(\mathsf{Opt}, n) &= \Omega\left(\frac{1}{\lg r}\lg\frac{\lg m}{2-\lg\eta_0}\right)\\ &= \Omega\left(\lg\frac{\lg n + O(1)}{2+O(1)}\right)\\ &= \Omega(\lg\lg n).\end{aligned}$$

- *Theorem* 3.5. Setting $r = 2$ and $m = n^{1+1/\lambda}$, λ fixed, we have $-\lg \eta_0 = \lg n/2\lambda + o(1)$ and hence

$$\begin{aligned}\overline{\textsc{Depth}}(\mathsf{Opt}, n) &= \Omega\left(\frac{1}{\lg r} \lg \frac{\lg m}{2 - \lg \eta_0}\right)\\ &= \Omega\left(\lg \frac{(1 + 1/\lambda) \lg n}{2 + \lg n/2\lambda + o(1)}\right)\\ &= \Omega\left(\lg \frac{1 + 1/\lambda}{1/2\lambda + o(1)}\right)\\ &= \Omega(\lg \lambda).\end{aligned}$$

- *Theorem* 3.7(b). Setting $r = \lg n/\lg\lg n$ and $m = O(n)$, we have $-\lg \eta_0 = r \lg r + O(1)$ and hence

$$\begin{aligned}\overline{\textsc{Depth}_r}(\mathsf{Opt}, n) &= \Omega\left(\frac{1}{\lg r} \lg \frac{\lg m}{2 - \lg \eta_0}\right)\\ &= \Omega\left(\frac{1}{\lg\lg\lg n} \lg \frac{\lg n + O(1)}{2 + r \lg r + O(1)}\right)\\ &= \Omega\left(\frac{\lg\lg n}{\lg\lg\lg n}\right).\end{aligned}$$

- *Theorem* 3.8. Setting $r = \lg \lambda/\lg\lg \lambda$, $m = n^{1+1/\lambda}$, λ fixed, we have $-\lg \eta_0 = -(r - 1)/\lambda \lg n + o(1))$ and hence

$$\begin{aligned}\overline{\textsc{Depth}_r}(\mathsf{Opt}, n) &= \Omega\left(\frac{1}{\lg r} \lg \frac{\lg m}{2 - \lg \eta_0}\right)\\ &= \Omega\left(\frac{1}{\lg\lg \lambda} \lg \frac{(1 + 1/\lambda) \lg n}{2 + (r - 1) \lg n/\lambda}\right)\\ &= \Omega\left(\frac{1}{\lg\lg \lambda} \lg \frac{(1 + 1/\lambda)}{(r - 1)/\lambda}\right)\\ &= \Omega\left(\frac{1}{\lg\lg \lambda} \lg \frac{\lambda + 1}{r - 1}\right)\\ &= \Omega\left(\frac{\lg\lg \lambda}{\lg\lg\lg \lambda}\right).\end{aligned}$$

4.3. The parallel variant. The memory-allocation scheme as used by Opt is slightly different here. Many hash functions can be applied in parallel to the same set. In an iteration t, let $m_{i,1}, m_{i,2}, \ldots$ be the cardinalities of ranges of those functions for some subset S_i, and let $m_i = m_{i,1} + m_{i,2} + \cdots$ be the total range used for it. The probability that all those hash functions will be a complete failure is

$$\prod_j m_{i,j}^{1-r_t}.$$

This probability is minimized when $m_{i,j} = 2$ for all j. In this case, the complete-failure probability is

$$2^{m_i(1-r_t)/2}.$$

Note that if the set size is greater than 2, then a complete separation is not possible if only two cells are allocated to a set. This does not pose a problem in our lower-bound analysis since we consider any partial separation to be a complete separation.

Let $m = m_1 + m_2 + \cdots + m_{N_t}$ be a memory allocation of the m cells to the N_t sets. The expected value of N_{t+1} is

$$\mathbf{E}\left(N_{t+1}\right) = \sum_{i=1}^{N_t} 2^{m_i(1-r_t)}.$$

Once again, this is minimized when all the m_i are equal. Hence we have the following result.

LEMMA 4.6. *In the parallel variant, all hash functions used by* Opt *are to a range of size* 2. *In iteration t, $m/2N_t$ functions with a total range of size m/N_t are applied to each one of the N_t sets of size r_t.*

From Lemma 4.6, we get a recursion formula for N_t:

$$\mathbf{E}\left(N_{t+1}\right) = N_t 2^{m(1-r_t)/2N_t}.$$

Note that Lemma 4.4 also holds here, so we can write

$$N_{t+1} \geq \frac{N_t}{4} 2^{m(1-r_t)/2N_t},$$

which will take a simpler form using the definition $\nu_t = m/N_t$:

$$\nu_{t+1} = 4\nu_t 2^{(r_t-1)\nu_t/2} \leq 2^{(r_t-1)\nu_t/2+\lg \nu_t+2}.$$

For $r_t \geq 4$, we have

$$\nu_{t+1} \leq 2^{r_t \nu_t}.$$

By setting $r_t = 4$ and $m = O(n)$, we get that $\nu_0 = O(1)$. The number of iterations T required to decrease the number of subsets below $\lg n$ (i.e., until $\nu_T = n/\lg n$) is $\Omega(\lg^* n)$. This completes the proof of the lower-bound part of Theorem 3.9.

The proof of (the chaining-variant part of) Theorem 3.10 is conducted in a similar manner to the lower-bound proof for the ordinary chaining variant. Let $r = r_1 = \lg^* n$, $r_{t+1} = r_t - 1$, and $m = O(n)$. Then by inequality (2),

$$\nu_0 = O\left((\lg^* n)^{\lg^* n}\right).$$

We can also write

$$\nu_{t+1} \leq 2^{r_t \nu_t} \leq 2^{\nu_t^2}.$$

Now the number of iterations required to achieve $\nu > \lg n$ is at least

$$\frac{\lg^* n - 1 - \lg^* \nu_0}{2} = \Omega(\lg^* n).$$

4.4. The retries variant. To complete the lower-bound analysis, we need to discuss the retries variant and provide proofs for Theorem 3.2 and items (a) and (c) of Theorem 3.7. Theorems 3.5 and 3.10 reference the retries variant as well. These references will be treated in a similar manner.

The retries technique is useful if a certain application of a hash function was not satisfactory according to some criteria. Then instead of coping with it, the algorithm may try another hash function. However, our simplifications allow Opt a dichotomous classification of poll results. If there was a complete failure in a hash of a specific internal node, then doing a retry is the same as what Opt will do in the next iteration, but with less memory. On the other hand, if this internal node was not a complete failure, we allow Opt to ignore it without any further resource investment. If retry was done on such a node then this may only result in a deeper tree. Retries cannot help in the root node either since the root node behavior is dominant (inequality (2)), and even $O(\lg n)$ retries in the root node will not yield a significantly better value for $N_0(r)$.

Thus the addition of retries to any model combination will not affect the DEPTH lower bounds. Upon further examination of the lower-bound proof of the chaining variant, it can be seen that a parallel insertion time of $O(\lg\lg n/\lg\lg\lg n)$ cannot be achieved unless a key is left in $\Omega(\lg\lg n/\lg\lg\lg n)$ nodes, which will nullify the ability of the retries variant to achieve $\text{SEARCH} = O(1)$.

The equivalence of an algorithm without retries to an algorithm with retries was possible here because the Opt could use its total memory allowance in each and every iteration. In general, using this technique to transform Alg, an algorithm that uses retries into Alg$'$, an algorithm that avoids retries leads to an increase in the memory used by the algorithm by a factor of up to $\text{DEPTH}(\mathsf{Alg}, n)$.

5. Proofs of upper bounds. As explained earlier, the main strategic decision made by a hashing algorithm is the allocation of memory to buckets. In the FKS algorithm, all hashing attempts (retries) of a bucket are into memory of fixed size. This scheme leads to $\text{DEPTH} = \Omega(\lg n)$. In order to further decrease the total number of hashing iterations, a more flexible memory allocation must be employed.

The idealized algorithms Opt used the same memory size in all iterations. The upper-bound algorithms in all model variants try to follow that scheme and use *almost* the same size of memory in every iteration. Thus in every iteration, the memory portion allocated to remaining buckets is increased.

A decreasing geometric series defines the partitioning of the total memory allowance between the iterations. Informally, we can say that although this series decreases quite rapidly, the decrease in the number of remaining buckets is so much quicker that the algorithm will find itself in conditions which are very similar to the extra-memory assumption which Opt was permitted to make.

The rest of this section is outlined as follows. We begin by reviewing the FKS algorithm. Next, we describe how this algorithm is modified to form our main algorithm, RetryShallow, which works in the retries model. We proceed by presenting minimal changes to RetryShallow, obtaining algorithm BasicShallow for the basic model. Algorithm ParShallow for the parallel model and ChainShallow for the chaining model are presented next.

5.1. Foundation: Algorithm FKS. Algorithm FKS [10] takes an arbitrary set $S \subseteq U$ of size n as input and in $O(n)$ expected time generates a hash table for S. The resulting table uses $O(n)$ storage and supports $O(1)$ lookup time. The algorithm builds a *two-level* hash table: a *level*-1 function splits S into subsets whose sizes are distributed in a favorable way; then an injective *level*-2 hash function is built for each subset.

The FKS algorithm assumes that $U = \{0, 1, \ldots, q-1\}$, where q is prime. This assumption does not lead to any loss of generality and we will adhere to it henceforth.

DEFINITION 5.1 (polynomial hash functions). *The class of d-degree polynomial hash functions is*

$$\mathcal{H}^d(m) := \left\{ h \;\middle|\; h(x) := 1 + \left(\sum_{i=0}^{d} a_i x^i \bmod q \right) \bmod m,\ a_i \in U \right\}.$$

The hash functions used in both levels of FKS are drawn from the class of linear (i.e., polynomial-of-degree-1) hash functions. The main properties of the linear hash functions which enable the construction are given in the following facts, proved in [10].

FACT 2. *Let S be fixed, and let h be chosen uniformly at random from the class $\mathcal{H}^1(|S|)$. Then*

$$\mathbf{E}\left(\sum_{i=1}^{m} |S_i|^2 \right) \leq 2.5|S|,$$

and, consequently (by Markov's inequality),

$$\mathrm{Prob}\left(\sum_{i=1}^{m} |S_i|^2 \leq 5|S| \right) \geq \frac{1}{2}.$$

FACT 3. *Let S be fixed, and let h be chosen uniformly at random from the class $\mathcal{H}^1(m)$. Then*

$$\mathrm{Prob}\,(h \textit{ is not injective on } S) \leq \frac{|S|^2}{m}.$$

The algorithm works in two phases as follows:

Phase I. For the given input set S, a level-1 hash function h is selected which satisfies

$$\sum_{i=1}^{n} \left|S_i^h\right|^2 < 5n. \tag{3}$$

This is done by repeatedly trying functions chosen at random from the class $\mathcal{H}^1(n)$. It follows from Fact 2 that the expected number of functions tried is constant. The memory used and the expected number of operations at this phase are therefore $O(n)$.

Phase II. For each bucket S_i, the algorithm allocates a range of size $2|S_i|^2$ and finds a level-2 function which injectively maps the bucket to this range. This function is constructed by repeatedly trying functions selected at random from the class $\mathcal{H}^1(2|S_i|^2)$. Each such function is noninjective with probability at most 1/2 (Fact 3). The expected number of functions tried is therefore constant. The memory usage and the expected number of operations per bucket are therefore quadratic in the size of the bucket. Since the algorithm insisted on attaining inequality (3) at Phase I, we have that the memory used and the expected number of operations at this phase are therefore $O(n)$ as well.

Viewing the algorithm within the tree-model framework, we see that the number of nodes ("active" buckets) drops by only a factor of approximately 2 from one level to the following one. This is the reason why the FKS algorithm has the order of $\lg n$ levels.

5.2. The retries variant: Algorithm RetryShallow. Algorithm RetryShallow is modeled after FKS. Phase I is identical and is concluded by finding a level-1 function h satisfying inequality (3). The main change in Phase II is that RetryShallow finds an injective hash function for all buckets by trying memory blocks of rapidly growing size and not of constant size as in the FKS algorithm. The block-size growth rate is characterized by the sequence $\{\beta_{t+1}\}$,

$$\beta_1 = 16, \qquad \beta_{t+1} = \frac{\beta_t^2}{4}, \tag{4}$$

or, in an explicit form,

$$\beta_t = 2^{2^t-2}. \tag{5}$$

Algorithm RetryShallow executes procedures $\mathsf{RetryShallow}_1, \ldots, \mathsf{RetryShallow}_{\lceil \lg n \rceil}$ simultaneously. A procedure $\mathsf{RetryShallow}_j$, $j = 1, \ldots, \lceil \lg n \rceil$, handles buckets S_i for which

$$2^{j-1} < |S_i| \leq 2^j.$$

Initially, all of these buckets are active. The procedure is a series of iterations in which buckets are deactivated by finding an injective hash function for them. Let $N_{t,j}$ be the number of keys in buckets belonging to procedure j which are active at the beginning of iteration t. Iteration t is a series of *attempts* to reduce $N_{t,j}$ by a factor of $\beta_t/2$. In each such attempt, each of the active buckets is hashed into a memory block of size $\beta_t 2^{2j}$ using a hash function selected at random from the class $\mathcal{H}^1(\beta_t 2^{2j})$. If the attempt fails to reduce the number of keys in active buckets to $2N_{t,j}/\beta_t$, then a "retry" is done in all nodes corresponding to active buckets; all separations obtained are disregarded. Otherwise, buckets for which an injective hash function was found become inactive; noninjective hash functions are disregarded (a retry) and the procedure carries on to iteration $t+1$.

The procedure terminates when there are no more active buckets belonging to it. The algorithm terminates when all procedures terminate.

Analysis of $\overline{\text{DEPTH}}(\mathsf{RetryShallow}, n)$. As noted in the description of FKS, the contribution of Phase I to $\overline{\text{DEPTH}}(\mathsf{RetryShallow}, n)$ is at most constant.

If $\beta_t \geq 4n$, then by the end of iteration t, the number of keys in active buckets is

$$\frac{2N_{t,j}}{\beta_t} \leq \frac{2n}{4n} = \frac{1}{2},$$

i.e., $N_{t+1,j} = 0$. It follows from (5) that setting $t = O(\lg\lg n)$ suffices to ensure $\beta_t \geq 4n$. Hence for all j, the number of iterations of $\mathsf{RetryShallow}_j$ is $O(\lg\lg n)$.

It follows from Fact 3 that a function selected at random from $\mathcal{H}^1(\beta_t 2^{2j})$ is not injective on a bucket of size at most 2^j with probability at most $1/\beta_t$. Hence the expected number of active keys by the end of an attempt is at most $N_{t,j}/\beta_t$. By Markov's inequality, the probability of any given attempt to be successful is at least

1/2. The expected number of attempts in an iteration is therefore at most 2 and the total expected number of attempts in any single procedure is $O(\lg\lg n)$.

Attempt failures are independent. We can therefore apply Chernoff bounds, obtaining that there is a constant C such that for any $\gamma > C$, the probability that the total number of attempts will be more than γ times its expected value is $o(1/\lg^{\gamma/C} n)$ in any given procedure. Since there are at most $\lceil \lg n \rceil$ procedures executing simultaneously, the expected parallel time until the slowest procedure terminates is also $O(\lg\lg n)$. We therefore have $\overline{\textsc{Depth}}(\mathsf{RetryShallow}, n) = O(\lg\lg n)$.

Analysis of $\textsc{Space}(\mathsf{RetryShallow}, n)$. As noted in the description of FKS, the contribution of Phase I to $\textsc{Space}(\mathsf{RetryShallow}, n)$ is $O(n)$. Consider any procedure $\mathsf{RetryShallow}_j$. In iteration t, there are at most $N_{t,j}/2^j$ active buckets; each such bucket is hashed into a memory block of size $\beta_t 2^{2j}$. The total memory used in the iteration is therefore bounded above by $N_{t,j}\beta_t 2^j$. (Recall that no space is charged for retry nodes.) The memory used in the following iteration is at most half of that,

$$N_{t+1,j}\beta_{t+1}2^j \le \frac{2N_{t,j}}{\beta_t}\cdot\frac{\beta_t^2}{4}2^j = \frac{N_{t,j}\beta_t 2^j}{2}.$$

Thus the total memory usage is, up to a constant factor, the same as memory used in iteration 1. Since initially $\mathbf{E}\left(\sum |S_i|\right) = n$ and $\beta_1 = O(1)$, we get that, even accounting for the possible doubling of set sizes due to rounding, $\textsc{Space}(\mathsf{RetryShallow}, n) = O(n)$.

Analysis of $\overline{\textsc{Time}}(\mathsf{RetryShallow}, n)$. Note that the expected number of times RetryShallow accesses any memory cell is constant. Then $\overline{\textsc{Time}}(\mathsf{RetryShallow}, n) = O(n)$ follows from $\textsc{Space}(\mathsf{RetryShallow}, n) = O(n)$. Since all functions used in RetryShallow are polynomials of degree 1, we get that the number of operations is linear even if arithmetic operations are counted and word size is limited to $O(\lg |U|)$.

Analysis of $\textsc{Search}(\mathsf{RetryShallow}, n)$. RetryShallow constructs a 2-level hash table for S. The first level consists of the function selected at Phase I which splits S to buckets. The second level consists of the injective hash functions found in Phase II for each of the buckets. We therefore have $\textsc{Search}(\mathsf{RetryShallow}, n) = 2$. This completes the proof of Theorem 3.3.

5.3. The basic model: Algorithm BasicShallow. Algorithm BasicShallow which works in the basic model is derived from RetryShallow by replacing *retry* nodes in RetryShallow by *refining* nodes. A description of the modifications to the algorithm and to the analysis follows.

Phase I. The algorithm creates an initial partition by selecting a hash function uniformly and at random from the class $\mathcal{H}^1(n)$. The goal of this phase is to achieve a partitioning of S into buckets $S_1, S_2, \ldots, S_n$ (some empty) such that inequality (3) holds. While this goal is not achieved, the algorithm iterates as follows. Each nonempty bucket S_i is hashed using a function selected at random from $\mathcal{H}^1(|S_i|)$. Since the total memory used for all buckets is n, this hashing forms a refined partition of S into n buckets. Let $S_1, S_2, \ldots, S_n$ now denote these newly created subsets of S. If inequality (3), is attained, then the phase ends; otherwise, the algorithm iterates again hashing each nonempty subset into a range equal to its size using a linear hash function.

Phase II. Similarly to Algorithm RetryShallow, Algorithm BasicShallow splits into procedures $\mathsf{BasicShallow}_1, \ldots, \mathsf{BasicShallow}_{\lceil \lg n \rceil}$. Consider a bucket S_i formed in Phase I, $2^{j-1} < |S_i| \le 2^j$; then S_i and all subbuckets formed from it during Phase II are handled by procedure $\mathsf{BasicShallow}_j$. Buckets and keys do not move between procedures.

Attempts of RetryShallow are translated to refining *rounds* of BasicShallow. Let $N_{t,j}$ be the number of keys in active buckets at the beginning of iteration $t \geq 1$ of the procedure $\mathsf{BasicShallow}_j$. Similarly to RetryShallow, the iteration ends when its rounds achieve that the number of keys in its active buckets is no greater than $2N_{t,j}/\beta_t$, where β_t is as before. In a round, a bucket of size r is hashed into a memory block of size $\beta_t r 2^j$ using a function selected at random from $\mathcal{H}^1(\beta_t r 2^j)$. If this function is injective, then the bucket becomes inactive. Otherwise, the bucket is split into smaller subbuckets and processing continues for all subbuckets which have two or more keys.

Analysis. It follows from the linearity of the expectation that Fact 2 holds also for each refining iteration of Phase I. The expected number of iterations in this phase is therefore $O(1)$. The contribution of the phase to the expected depth is constant and its contribution to the expected memory usage is $O(n)$. The analysis of Phase II follows.

Consider a round of iteration t of $\mathsf{BasicShallow}_j$. The probability for a bucket of size r, active at the beginning of the round, to remain active after the round is at most $r/(2^j\beta_t) < 1/\beta_t$. As before, the expected number of active keys by the end of the round is at most $N_{t,j}/\beta_t$; the probability that the round will fail (i.e., that it will not terminate the iteration) is at most 1/2; the expected number of rounds in an iteration is therefore at most 2. Following along the lines of the analysis of RetryShallow, we have $\overline{\text{DEPTH}}(\mathsf{BasicShallow}, n) = O(\lg\lg n)$ and therefore $\overline{\text{SEARCH}}(\mathsf{BasicShallow}, n) = O(\lg\lg n)$.

The memory-allocation scheme is such that memory allocated to all subbuckets formed from a certain bucket is never greater than what would have been allocated to the bucket had it not been split. Memory cannot be reused in the basic model, so even nonsuccessful rounds contribute to the total memory usage. However, we have that the memory used in rounds of the same iteration is essentially the same and that the expected number of rounds in an iteration is $O(1)$. It follows that the expected memory used in an iteration of BasicShallow is no more than a constant times the memory used of the same iteration of RetryShallow. This completes the proof of Theorem 3.4.

5.4. The parallel variant: Algorithm ParShallow. Both RetryShallow and BasicShallow can be implemented on a parallel machine where each processor is initially assigned a key and keys do not move among processors [12]. The ParShallow algorithm described below achieves $O(\lg^* n)$ depth in the parallel model by applying several hash functions to a key in each tree node. In a parallel setting, this corresponds to allowing multiple processors to hash the same key. In each successive iteration, more and more processors are drafted to hash fewer and fewer keys. The bookkeeping required in each iteration for identifying the active keys and assigning processors to them is not trivial. A PRAM algorithm running in $O(\lg^* n)$ time was first described by Matias and Vishkin [22]. Their algorithm is based in part on ParShallow, which we describe next. We use the description and the analysis of RetryShallow as a skeleton.

The following differences apply. The sequence β_t starts with $\beta_1 = 4$ and increases at a quicker rate, $\beta_t = 2^{\beta_t - 2}$. In iteration t of procedure $\mathsf{ParShallow}_j$ the $\beta_t r 2^j$-size memory block allocated to a bucket of $2^{j-1} < r \leq 2^j$ keys is further divided into $\beta_t 2^j/2r$ subblocks of $2r^2$ cells each. The parallel hashings are then done into these subblocks. The probability of failure in injectively mapping a bucket of size r into a subblock of size r is at most 1/2. Therefore, the probability to fail in all $\beta_t 2^j/2r$ simultaneous trials is at most

$$2^{-\beta_t 2^j/2r} < 2^{-\beta_t/2}.$$

In iteration t of the jth procedure, rounds continue until the number of active keys is at most $2 \cdot 2^{-\beta_t/2} N_{t,j}$, and hence

$$N_{t+1,j} \leq 2 \cdot 2^{-\beta_t/2} N_{t,j}.$$

This last bound together with the definition of the sequence $\{\beta_t\}$ proves that $\overline{\text{DEPTH}}(\text{ParShallow}) = O(\lg^* n)$. To see that $\overline{\text{SPACE}}(\text{ParShallow}) = \text{SPACE}_r(\text{ParShallow}) = O(n)$, note that the total memory used by a round of iteration $t+1$ is at most $1/2$ of the total memory used by a round of iteration t and that the total memory used in the first iteration is linear. This completes the proof of the upper-bound part of Theorem 3.9.

5.5. The chaining variant: Algorithm ChainShallow. The reduction in DEPTH to $O(\lg\lg n/\lg\lg\lg n)$ is achieved by replacing Phase II of RetryShallow by two phases:

Phase IIa. Use the prototype of algorithm RetryShallow to continuously break subsets until they are all small enough instead of attempting to achieve a complete separation. In particular, hashing is conducted until all subsets have $R = \lg\lg n/\lg\lg\lg n$ or fewer keys.

Phase IIb. When all subsets are smaller than R, then in each tree level, hashing is into a range of size 1. Recall that the chaining-model variant allows for storing a key in every internal node. At most R levels will thus be added to the tree regardless of the hash function employed.

Phase IIa of ChainShallow therefore deals with sets of at least R elements; as soon as a subset is broken into smaller pieces, the phase ceases to handle it. We need the hashing functions used to satisfy a "good-breaking" property.

DEFINITION 5.2. *Let $\mathcal{H}$ be a class of hash functions mapping sets of size r into memory of size βr^2. Let h be picked at random from $\mathcal{H}$. Then $\mathcal{H}$ is a* good breaker *iff*

$$\text{Prob}\left(\forall i \left|h^{-1}(i)\right| \leq R\right) \leq \beta^{1-R}.$$

This property is achieved by random functions [11, Chap. 2]. However, true random functions are not useful for hashing since they require huge space for representation and consequently nonconstant evaluation time. The class of polynomial hash functions of degree R is a good breaker as well. Its members can be represented efficiently. Unfortunately, each application of a hash function of this class requires the order of R steps, which amounts to a total of $O(\lg\lg n)$ time (although the number of hash function applications is still $O(\lg\lg n/\lg\lg\lg n)$).

Dietzfelbinger and Meyer auf der Heide [9] gave a construction of a good-breaker class $\mathcal{R}$ such that a function $h \in \mathcal{R}$ can be evaluated in constant time. This class $\mathcal{R}$ is best suited for a more practical implementation of ChainShallow since it offers the advantages of sublinear representation and constant-time evaluation.

The sequence $\{\beta_t\}$ is defined in the algorithm ChainShallow by

$$\beta_{t+1} = \frac{\beta_t^R}{4}, \qquad \beta_1 = O(1). \tag{6}$$

As in RetryShallow, the basic unit of memory allocation is $\beta_t 2^{2j}$ in iteration t of ChainShallow$_j$. A subset of size $r \leq 2^j$ belonging to this procedure is hashed using a good breaker into a $\beta_t r 2^j$-size memory block. The probability that the subset will not be broken into small enough subsets is at most β_t^{1-R}. The rounds of an iteration carry on until $N(r) \leq 2N_t(r)\beta_t^{1-R}$; then the expected number of rounds in an iteration will be ≤ 2 here as well. Thus we have

$$N_{t+1}(r) \leq 2N_t(r)\beta_t^{1-R}. \tag{7}$$

Combining recurrences (6) and (7), we see that memory requirements decrease geometrically:

$$N_{t+1}(r)\beta_{t+1} \leq \frac{N_t(r)\beta_t}{2}.$$

Adding to this the fact that the root-node function must have satisfied inequality (3), we infer that the total memory usage is linear.

Solving recurrence (6), we have

$$\beta_t = \beta_1^{R^{t-1}} 4^{-(R^t-1)/(R-1)},$$

from which it follows that the number of iterations is $O(R)$, completing the proof of Theorem 3.6.

6. Concluding remarks. The hashing model proposed leads to an alternate view of hashing algorithms as element-distinctness-proof generators. In our lower-bound analysis, we assumed that all of the hash functions (the proof components) are completely random. However, it is not difficult to see that if the universe is not too large, say polynomial in the size of the input set, and if nonrandom functions can be used, then the lower bounds do not hold (e.g., by an integer-sorting algorithm).

Is the random-functions assumption essential? It was shown before [1, 31] that random functions are optimal in certain hashing situations. On the other hand, hashing algorithms that are based on open addressing or on the FKS scheme (such as the one described in [13]) as well as the upper bounds presented here do not assume the existence of random functions. We conjecture that the lower-bound results hold for a superpolynomial-sized universe even if arbitrary functions are allowed.

Although the model proposed in this paper is very general, it does not cover all hashing algorithms (e.g., the one presented in [9]). It may be interesting to define more general models and to gain better understanding of these models as well.

Acknowledgment. Fruitful comments made by Faith E. Fich are gratefully acknowledged.

REFERENCES

[1] M. Ajtai, J. Komlós, and E. Szemerédi, *There is no fast single hashing function*, Inform. Process. Lett., 7 (1978), pp. 270–273.

[2] N. Alon and J. H. Spencer, *The Probabilistic Method*, John Wiley, New York, 1991.

[3] D. Angluin and L. G. Valiant, *Fast probabilistic algorithms for hamiltonian paths and matchings*, J. Comput. System. Sci., 18 (1979), pp. 155–193.

[4] W. C. Chen and J. S. Vitter, *Analysis of early-insertion standard coalesced hashing*, SIAM J. Comput., 12 (1983), pp. 667–676.

[5] ———, *Analysis of new variants of coalesced hashing*, ACM Trans. Database Systems, 9 (1984), pp. 616–645.

[6] ———, *Deletion algorithms for coalesced hashing*, Comput. J., 29 (1986), pp. 436–450.

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

[8] M. Dietzfelbinger, A. R. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, and R. E. Tarjan, *Dynamic perfect hashing: Upper and lower bounds*, SIAM J. Comput., 23 (1994), pp. 738–761.

[9] M. Dietzfelbinger and F. Meyer auf der Heide, *A new universal class of hash functions and dynamic hashing in real time*, in Proc. 1990 International Colloquium on Automata, Languages, and Programming, Springer-Verlag, Berlin, New York, Heidelberg, 1990, pp. 6–19.

[10] M. L. Fredman, J. Komlós, and E. Szemerédi, *Storing a sparse table with $O(1)$ worst case access time*, J. Assoc. Comput. Mach., 31 (1984), pp. 538–544.

[11] J. Gil, *Lower bounds and algorithms for hashing and parallel processing*, Ph.D. thesis, Hebrew University of Jerusalem, Jerusalem, 1990.

[12] J. Gil and Y. Matias, *Fast and efficient simulations among CRCW models*, J. Parallel Distrib. Comput., 23 (1994), pp. 135–148.

[13] ———, *Fast hashing on a PRAM: Designing by expectation*, in Proc. 1991 Symposium on Discrete Algorithms, Society for Industrial and Applied Mathematics, Philadelphia, 1991, pp. 271–280.

[14] J. Gil, F. Meyer auf der Heide, and A. Wigderson, *Not all keys can be hashed in constant time*, in Proc. 1990 ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1990, pp. 244–253.

[15] T. Hagerup and C. Rüb, *A guided tour of chernoff bounds*, Inform. Process. Lett., 33 (1989/1990), pp. 305–308.

[16] N. L. Johnson and S. Kotz, *Urn Models and their Application*, John Wiley, New York, 1977.

[17] A. R. Karlin and E. Upfal, *Parallel hashing: An efficient implementation of shared memory*, in Proc. 1986 ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1986, pp. 160–168.

[18] G. D. Knott, *Direct chaining with coalescing lists*, J. Algorithms, 4 (1984), pp. 7–21.

[19] D. E. Knuth, *Sorting and Searching*, The Art of Computer Programming, vol. 3, Addison–Wesley, Reading, MA, 1973.

[20] ———, *Computer science and its relationship to mathematics*, Amer. Math. Monthly, 8 (1974), pp. 323–343.

[21] Y. Matias, personal communication, 1990.

[22] Y. Matias and U. Vishkin, *Converting high probability into nearly-constant time: With applications to parallel hashing*, in Proc. 1991 ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1991, pp. 307–316; Technical report UMIACS-TR-91-65, Institute for Advanced Computer Studies, University of Maryland, College Park, MD, 1991.

[23] K. Mehlhorn, *Data Structures and Algorithms* I: *Sorting and Searching*, EATCS Monographs on Theoretical Computer Science, Springer-Verlag, Berlin, Heidelberg, 1984.

[24] B. K. Pittel, *Linear probing the probable largest search time grows logarithmically with the number of records*, J. Algorithms, 8 (1987), pp. 236–249.

[25] B. K. Pittel and J.-H. Yu, *On search times for early-insertion coalesced hashing*, SIAM J. Comput., 17 (1988), pp. 492–503.

[26] M. Sipser, *A complexity theoretic approach to randomness*, in Proc. 1983 ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1983, pp. 330–335.

[27] L. J. Stockmeyer, *The complexity of approximate counting*, in Proc. 1983 ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1983, pp. 118–126.

[28] R. E. Tarjan and A. C. Yao, *Storing a sparse table*, Comm. Assoc. Comput. Mach., 21 (1979), pp. 606–611.

[29] J. S. Vitter, *Analysis of coalesced hashing*, Ph.D. thesis, Technical report STAN-CS-80-817, Department of Computer Science, Stanford University, Stanford, CA, 1982.

[30] A. C. Yao, *Should tables be sorted?*, J. Assoc. Comput. Mach., 28 (1981), pp. 615–628.

[31] ———, *Uniform hashing is optimal*, J. Assoc. Comput. Mach., 32 (1985), pp. 687–693.

SIAM J. COMPUT.
Vol. 25, No. 5, pp. 956–997, October 1996

ON-LINE PLANARITY TESTING[*]

GIUSEPPE DI BATTISTA[†] AND ROBERTO TAMASSIA[‡]

Abstract. The *on-line planarity-testing* problem consists of performing the following operations on a planar graph G: (i) testing if a new edge can be added to G so that the resulting graph is itself planar; (ii) adding vertices and edges such that planarity is preserved. An efficient technique for on-line planarity testing of a graph is presented that uses $O(n)$ space and supports tests and insertions of vertices and edges in $O(\log n)$ time, where n is the current number of vertices of G. The bounds for tests and vertex insertions are worst-case and the bound for edge insertions is amortized. We also present other applications of this technique to dynamic algorithms for planar graphs.

Key words. planar graph, on-line algorithm, dynamic algorithm

AMS subject classifications. 68R10, 05C10, 68Q20, 68P05

1. Introduction.

The problems of testing planarity and constructing planar embeddings of graphs have been extensively studied in the past years and find direct application in a variety of areas including circuit layout, graphics, computer-aided design, and automatic graph drawing.

In a static environment, where an n-vertex graph G is entirely known in advance, we can test the planarity of G and compute a planar embedding in optimal $O(n)$ time [5, 8, 18, 20, 31, 38]. In a dynamic environment, where a planar graph G is assembled on-line by insertions of vertices and edges, we would like to determine quickly whether an update causes G to become nonplanar. Namely, the *on-line planarity-testing* problem consists of performing the following operations on a planar graph G: (i) testing if a new edge can be added to G so that the resulting graph is itself planar; (ii) adding vertices and edges such that planarity is preserved.

While many research efforts have been focused on planar graphs and on dynamic graph algorithms, the development of an efficient algorithm for on-line planarity testing has been an elusive goal.

Recent results on planar graphs include algorithms for parallel planarity testing [37, 46], embedding [4, 62], drawing [10, 13, 19, 51], reachability [36, 54, 57], shortest paths [22], and minimum spanning trees [15, 21]. Previous work on dynamic graph algorithms is surveyed in §2. The technique of [53] is a first step toward on-line planarity testing. Namely, it solves the restricted problem of maintaining a planar embedding of a planar graph. It uses $O(n)$ space and supports queries (testing whether two vertices are on the same face of the embedding) and updates (adding vertices and edges to the embedding) in $O(\log n)$ time.

[*] Received by the editors November 2, 1994; accepted for publication (in revised form) January 24, 1995. This paper includes results presented at the 30*th IEEE Symposium on Foundations of Computer Science* (1989) and the 17*th International Colloquium on Automata, Languages, and Programming* (1990). This research was supported in part by the ESPRIT II Basic Research Actions Program of the EC under contract 3075 (project ALCOM), National Science Foundation grants CCR-9007851 and CCR-9423847, Office of Naval Research/Defense Advanced Research Projects Agency contract N00014-91-J-4052, ARPA order 8225, the Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo of the Italian National Research Council, and U. S. Army Research Office grant DAAL03-91-G-0035. This research performed in part while G. Di Battista was with the University of Rome "La Sapienza" and with the Universitá della Basiliata, while G. Di Battista was visiting Brown University, and while R. Tamassia was visiting the University of Rome "La Sapienza."

[†] Dipartimento di Discipline Scientifiche, Sezione Informatica, Terza Universitá degli Studi di Roma, 84 Via della Vasca Navale, Rome 00146, Italy (dibattista@iasi.rm.cnr.it).

[‡] Department of Computer Science, Brown University, Providence, RI 02912-1910 (rt@cs.brown.edu).

In this paper, a technique for on-line planarity testing is presented that uses $O(n)$ space and supports tests and updates in $O(\log n)$ time, n being the current number of vertices of the graph. The bounds for tests and vertex insertions are worst-case and the bound for edge insertions is amortized.

The following repertoire of query and update operations is defined for a planar graph G:

Test(v_1, v_2): Determine whether edge (v_1, v_2) can be added to G while preserving planarity, i.e., test whether graph G admits a planar embedding Γ such that v_1 and v_2 are on the boundary of the same face of Γ.

InsertEdge(e, v_1, v_2): Add edge e between vertices v_1 and v_2 to graph G. The operation is allowed only if the resulting graph is itself planar.

InsertVertex(e, v, e_1, e_2): Split edge e into two edges e_1 and e_2 by inserting vertex v.

AttachVertex(e, v, u): Add vertex v and connect it to vertex u by means of edge e.

MakeVertex(v): Add an isolated vertex v.

Our main result is expressed by the following theorem.

THEOREM 1.1. *Let G be a planar graph that is dynamically updated by adding vertices and edges, and let n be the current number of vertices of G. There exists a data structure for the on-line planarity-testing problem in G with the following performance: the space requirement is $O(n)$; operation MakeVertex takes worst-case time $O(1)$; operations Test, AttachVertex, and InsertVertex take worst-case time $O(\log n)$; and operation InsertEdge takes amortized time $O(\log n)$.*

The techniques developed in our work provide new insights on the topological properties of planar *st*-graphs and on the relationship between planarity and the decomposition of a graph into its biconnected and triconnected components.

The rest of this paper is organized as follows. In §2, we survey previous results on dynamic graph algorithms. Section 3 provides basic definitions. In §4, we present a static data structure that supports only operation *Test* in biconnected graphs. Sections 5 and 6 describe the dynamic data structure for on-line planarity testing in biconnected graphs. The data structure is extended to general planar graphs in §7. Finally, some applications of our technique to graph planarization, on-line transitive closure, and on-line minimum spanning trees are given in §8.

2. Dynamic graph algorithms. The development of dynamic algorithms for graph problems has acquired increasing theoretical interest, motivated by many important applications in network optimization, very large-scale integration (VLSI) layout, computational geometry, and distributed computing. In this section, we survey representative dynamic graph algorithms for reachability, shortest paths, minimum spanning trees, and connectivity. Throughout this section, n and m, respectively, denote the number of vertices and edges of the graph being considered. A general lower-bound technique for incremental algorithms, with applications to dynamic graph algorithms, is discussed in [3].

A reachability query in a digraph asks whether there is a directed path between two vertices. For general digraphs, there exist insertions-only semidynamic data structures with $O(n^2)$ space, $O(1)$ query time, and $O(n)$ amortized update time [6, 32, 43]. The same performance is achieved for deletions only in acyclic digraphs [6, 33]. Fully dynamic data structures with $O(n)$ space and $O(\log n)$ query and update time exist for some classes of planar digraphs [11, 34, 54, 56]. The related problem of maintaining a topological ordering of an acyclic digraph is studied in [1].

A shortest-path query in a digraph asks for the length of a shortest path between two vertices. Fully dynamic data structures for shortest-path queries are presented in [16, 48]. They have $O(n^2)$ space, $O(1)$ query time, $O(n^2)$ time for edge insertion, and $O(mn+n^2 \log n)$ time for edge deletion. The best-known semidynamic data structures supporting insertions in digraphs with unit edge lengths use $O(n^2)$ space and have constant query time; the total time to process all edge insertions is $O(n^3 \log n)$, which amortizes to $O(n \log n)$ time per insertion for dense graphs [2, 39]. For series-parallel digraphs with weighted edges, there exists a fully dynamic $O(n)$-space data structure that supports queries and updates in $O(\log n)$ time [9]. This data structure also maintains a maximum flow within the same time bounds.

The dynamic maintenance of minimum spanning trees has the interesting property that, after an update operation consisting of a weight change or adding/deleting an edge, at most one edge needs to be replaced in the minimum spanning tree. For general graphs, the best result is $O(\sqrt{m})$ update time and $O(m)$ space [21]. In the special case of planar graphs, updates can be done in $O(\log n)$ time using an $O(n)$-space fully dynamic data structure [11, 15].

Regarding connectivity problems, a classical result shows that the connected components of a graph can be efficiently maintained in a semidynamic environment where only edge-insertions are performed, by means of a union-find data structure [58]. A sequence of k queries and edge insertions takes time $O(k\alpha(k,n))$, where $\alpha(k,n)$ denotes the slowly growing inverse of Ackermann's function. The same performance is obtained for biconnected components [42, 60], triconnected components [11, 42], and four-connected components [35]. Semidynamic techniques supporting deletions only are studied in [17, 47]. In a fully dynamic environment, the connected components of a general graph can be maintained in time $O(\sqrt{m})$ per update operation [21], while the biconnected components of a planar graph can be maintained in $O(n^{2/3})$ time using $O(n)$ space [25]. The related problems of maintaining the two- and three-edge-connected components are studied in [23, 24, 60].

3. Preliminaries.

We assume that the reader is familiar with graph terminology and basic properties of planar graphs (see, e.g., [40]). Throughout this paper, n denotes the number of vertices of the planar graph G currently being considered. Unless otherwise specified, we only consider graphs without self-loops and multiple edges. Recall that a planar graph without self-loops and multiple edges has $O(n)$ edges.

First, we review some definitions on graph connectivity. A separating k-set of a graph G is a set of k vertices whose removal increases the number of connected components of G. Separating 1-sets and 2-sets are called *cutvertices* and *separation pairs*, respectively. A connected graph is said to be *biconnected* if it has no cutvertices. The *blocks* of a connected graph (also called *biconnected components*) are its maximal biconnected subgraphs. A graph is *triconnected* if it is biconnected and has no separation pairs.

A *planar drawing* of a graph is such that no two edges intersect (except possibly at the endpoints). A graph is *planar* if it admits a planar drawing. A planar drawing partitions the plane into topologically connected regions, called *faces*. The unbounded face is called the *external face*. The *boundary* of a face is its delimiting circuit. All the face boundaries of a biconnected graph are simple circuits. For brevity, we sometimes use "face" to mean "face boundary."

The *incidence list* of a vertex v is the set of edges incident upon v. A planar drawing determines a circular ordering on the incidence list of each vertex v according to the clockwise sequence of the incident edges around v.

Two planar drawings of the same connected graph G are *equivalent* if they determine the same circular orderings of the incidence lists. Two equivalent planar drawings have the same face boundaries. A *planar embedding* or simply *embedding* Γ of G is an equivalence class of planar drawings and is described by circularly sorted incidence lists for each vertex v. The face boundaries of any drawing of Γ are called the faces of Γ. A triconnected planar graph has a unique embedding, up to reversing all the incidence lists.

A *planar st-graph* G is a planar acyclic digraph with exactly one source (vertex without incoming edges) s and exactly one sink (vertex without outgoing edges) t which admits a planar embedding such that s and t are on the same face. Such graphs were first introduced in [38]. Vertices s and t are called the *poles* of G. Henceforth, we shall only consider embeddings of a planar st-graph such that s and t are on the same face. Following the developments of [10, 13], we visualize a planar st-graph with s and t on the external face and all the edges directed upward; see Fig. 1.

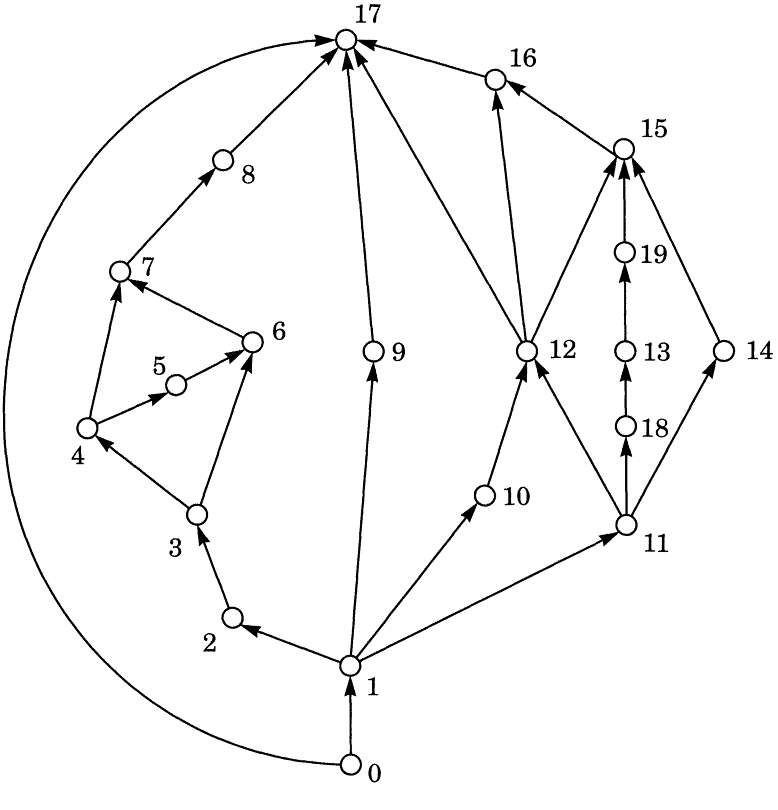


FIG. 1. *Example of a planar st-graph.*

The following properties are demonstrated in [55].

LEMMA 3.1. *Let Γ be a planar embedding of a planar st-graph G.*

1. *The incoming edges of each vertex v of G appear consecutively in the incidence list of v sorted according to Γ, and so do the outgoing edges.*

2. *Each face of Γ consists of the concatenation of two directed paths.*

With reference to the second property, the origin and destination of the paths forming a face f are called the *extreme vertices* of f. The other vertices of f are called *internal vertices* of f. For example, the planar st-graph of Fig. 1 has a face with extreme vertices 11 and 15 and with internal vertices 12, 18, 13, and 19.

A *planar st-orientation* of an undirected graph G is an orientation of the edges of G such that the resulting directed graph is a planar st-graph. A graph G admits a planar st-orientation if and only if it is *planarly st-biconnectible* [38], i.e., the graph obtained from G by adding the edge (s, t) is planar and biconnected. A planar st-orientation can be computed in $O(n)$ time [18].

4. Tests. In this section, we consider the problem of performing operation *Test* on a biconnected planar graph with n vertices. We assume that the graph has been oriented into a planar st-graph such that s and t are adjacent.

4.1. Decomposition tree. Let G be a planar st-graph. A *split pair* of G is either a separation pair or a pair of adjacent vertices. A *split component* of a split pair $\{u, v\}$ is either an edge (u, v) or a maximal subgraph C of G such that C is an uv-graph and $\{u, v\}$ is not a split pair of C. A *maximal split pair* $\{u, v\}$ of G is such that there is no other split pair $\{u', v'\}$ in G such that $\{u, v\}$ is contained in a split component of $\{u', v'\}$.

For example, in the planar st-graph G of Fig. 1, the pair $\{3, 7\}$ is a split pair but not a maximal split pair, while $\{0, 17\}$ is the only maximal split pair of G. The split components of the split pair $\{1, 17\}$ are shown in Fig. 2.

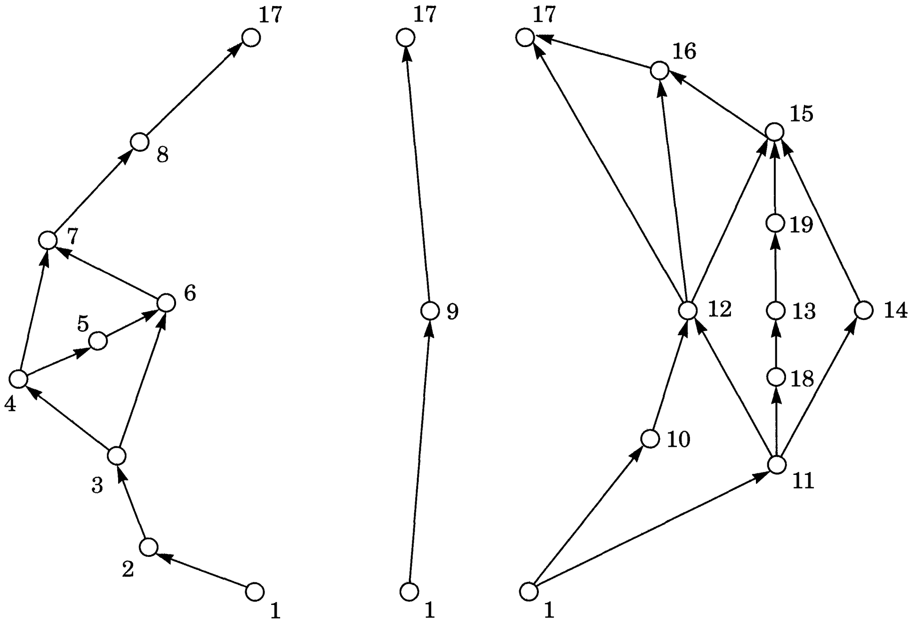


FIG. 2. *Split components of the split pair* $\{1, 17\}$ *in the planar st-graph of Fig.* 1.

The *decomposition tree* $\mathcal{T}$ of G describes a recursive decomposition of G with respect to its split pairs and will be used to synthetically represent all the embeddings of G with vertices s and t on the external face. Tree $\mathcal{T}$ is a rooted ordered tree whose nodes are of four types: S, P, Q, and R. Each node μ of $\mathcal{T}$ has an associated planar st-graph (possibly with multiple edges), called the *skeleton* of μ and denoted by *skeleton*(μ). Also, it is associated with an edge of the skeleton of the parent ν of μ, called the *virtual edge* of μ in *skeleton*(ν). Tree $\mathcal{T}$ is recursively defined as follows.

Trivial case. If G consists of a single edge from s to t, then $\mathcal{T}$ consists of a single Q-node whose skeleton is G itself.

Series case. If G is not biconnected, let $c_1, \ldots, c_{k-1}$ ($k \geq 2$) be the cutvertices of G. Since G is planarly st-biconnectible, each cutvertex c_i is contained in exactly two blocks G_i and G_{i+1} such that s is in G_1 and t is in G_k. The root of $\mathcal{T}$ is an S-node μ. Graph *skeleton*(μ) consists of the chain $e_1, \ldots, e_k$, where edge e_i goes from c_{i-1} to c_i, $c_0 = s$, and $c_k = t$, plus the edge (s,t). (See Fig. 3(a))

Parallel case. If s and t are a split pair for G with split components $G_1, \ldots, G_k$ ($k \geq 2$), the root of $\mathcal{T}$ is a P-node μ. Graph *skeleton*(μ) consists of $k+1$ parallel edges from s to t, denoted $e_1, \ldots, e_{k+1}$. (See Fig. 3(b))

Rigid case. If none of the above cases applies, let $\{s_1, t_1\}, \ldots, \{s_k, t_k\}$ be the maximal split pairs of G ($k \geq 1$), and for $i = 1, \ldots, k$, let G_i be the union of all the split components of $\{s_i, t_i\}$. The root of $\mathcal{T}$ is an R-node μ. Graph *skeleton*(μ) is obtained from G by replacing each subgraph G_i with the edge e_i from s_i to t_i and by adding the edge (s,t). Notice that the skeleton of an R-node is triconnected. (See Fig. 3(c))

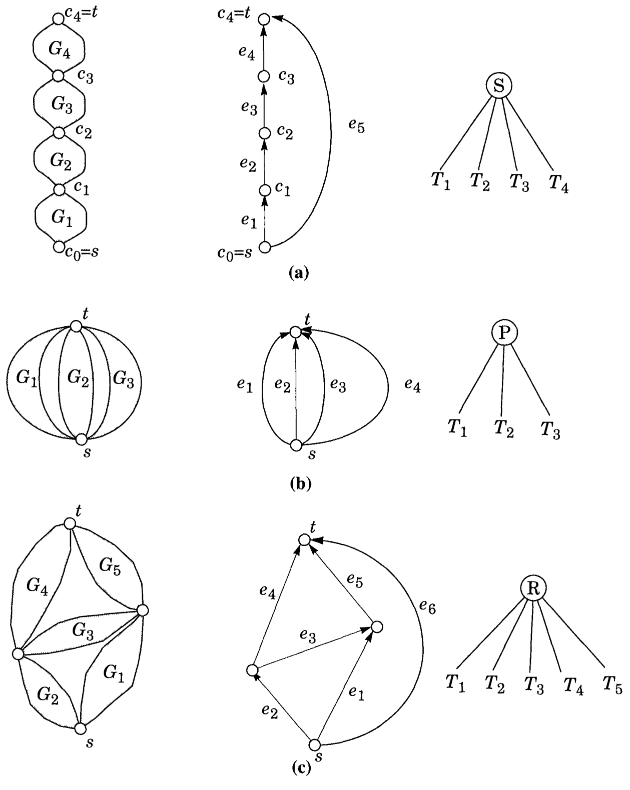


FIG. 3. (a) *Series decomposition.* (b) *Parallel decomposition.* (c) *Rigid decomposition.*

In the last three cases (series, parallel, and rigid), μ has children $\mu_1, \ldots, \mu_k$ (in this order), such that μ_i is the root of the decomposition tree of graph G_i ($i = 1, \ldots, k$). The virtual edge of node μ_i is edge e_i of *skeleton*(μ). Graph G_i is called the *pertinent graph* of node μ_i, and the *expansion graph* of e_i. (Note that G is the pertinent graph of the root.) We denote with s_μ and t_μ the poles of the skeleton of a node μ. We find it convenient (e.g., in Theorem 4.6 below) to define the expansion graph of a vertex of *skeleton*(μ) as the vertex itself.

Figure 4 illustrates the decomposition tree and the skeletons of the R-nodes for the planar st-graph of Fig. 1. Our definition of decomposition tree is a variation of the one given in [4] and is closely related to the decomposition of biconnected graphs into triconnected components [30].

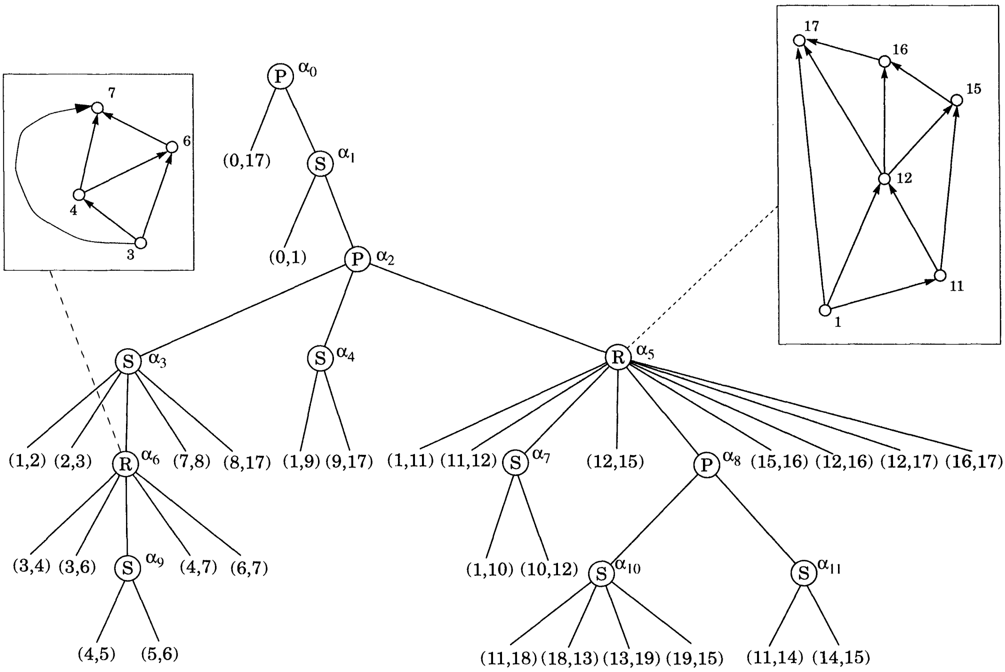


FIG. 4. *Decomposition tree $\mathcal{T}$ for the planar st-graph of Fig.* 1 *and skeletons of the R-nodes.*

LEMMA 4.1. *The decomposition tree $\mathcal{T}$ of G has $O(n)$ nodes. Also, the total number of edges of the skeletons stored at the nodes of $\mathcal{T}$ is $O(n)$.*

Proof. The leaves of $\mathcal{T}$ are Q-nodes in one-to-one correspondence with the edges of G, and each internal node of $\mathcal{T}$ has at least two children. Hence $\mathcal{T}$ has $O(n)$ nodes. If a node μ of $\mathcal{T}$ has k children, then *skeleton*(μ) has at most $k+1$ edges (one edge for a Q-node, k edges for an S- or P-node, and $k+1$ edges for an R-node). Hence the total number of edges of the skeletons is at most the sum of the number of nodes and edges of $\mathcal{T}$ and thus is $O(n)$. □

Now, we show how the decomposition tree can be used to represent all the planar embeddings of a planar st-graph G with the edge (s,t). Let Γ be a planar embedding of G.

Two basic primitives can be used to obtain a new planar embedding from Γ. A *reverse* operation consists of flipping a split component around its poles. A *swap* operation consists of exchanging the position of two split components of the same split pair. For example, Fig. 5 shows the planar embedding obtained from that of Fig. 1 by means of two swap operations and one flip operation.

LEMMA 4.2 (see [12]). *Given a pair of embeddings Γ' and Γ'' of a planar st-graph G with the edge (s,t), Γ'' can be obtained from Γ' by means of a sequence of $O(n)$ reverse and swap operations.*

By Lemma 4.2, the decomposition tree $\mathcal{T}$ can be used to represent an embedding

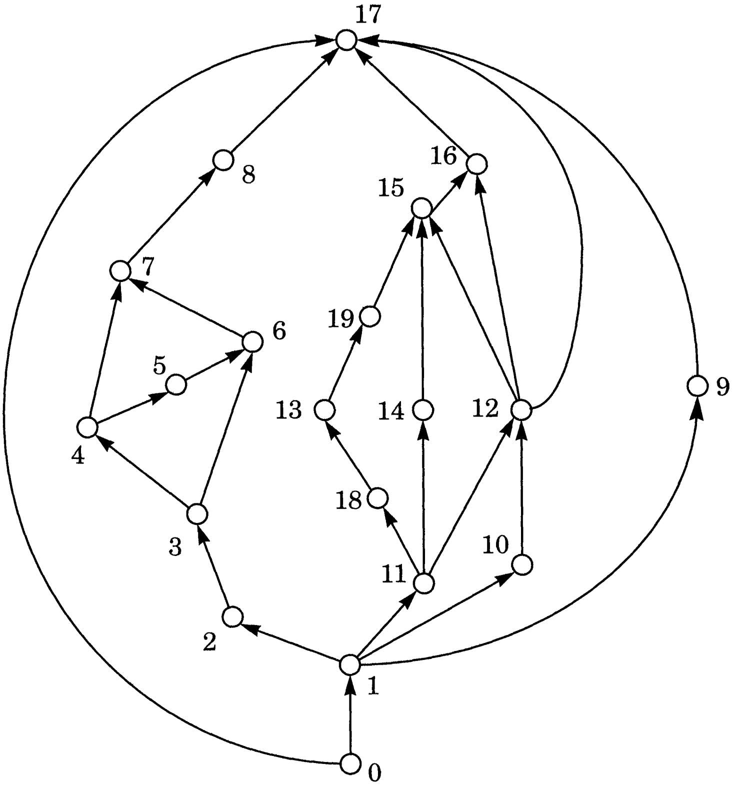


FIG. 5. *Embedding obtained from the one of Fig.* 1 *by means of two swap operations around the split pairs* (1, 17) *and* (11, 15) *and one flip operation around the split pair* (1, 17).

of G by

1. selecting one of the two possible flips of the skeleton of each R-node around its poles; and
2. selecting a permutation of the skeletons of the children of each P-node with respect to their common poles.

Before presenting the algorithm for operation *Test*, we need to introduce additional concepts.

Let v be a vertex of G. The *allocation nodes* of v are the nodes of $\mathcal{T}$ whose skeleton contains v. Note that v has at least one allocation node. For example, with reference to the planar st-graph of Fig. 1 and its decomposition tree shown in Fig. 4, we have that the allocation nodes of vertex 15 are the Q-nodes associated with its incident edges plus nodes α_5, α_8, α_{10}, and α_{11}.

The following facts can be easily proved.

FACT 1. *The pertinent graphs of the children of a node μ can only share vertices of skeleton(μ).*

FACT 2. *If v is in skeleton(μ), then v is also in the pertinent graph of all the ancestors of μ.*

FACT 3. *If v is a pole of skeleton(μ), then v is also in the skeleton of the parent of μ.*

FACT 4. *If v is in skeleton(μ) but is not a pole of skeleton(μ), then v is not in the skeleton of any ancestor of μ.*

LEMMA 4.3. *Let v be a vertex of G. The least common ancestor μ of the allocation nodes of v is itself an allocation node of v. Also, if $v \neq s, t$, then μ is the only allocation node of v such that v is not a pole of skeleton(μ).*

Proof. For the first part of the lemma, it is sufficient to show that the least common ancestor μ of two allocation nodes μ_1 and μ_2 of v is itself an allocation node of v. This is trivial if one of μ_1 and μ_2 is an ancestor of the other. Otherwise, by Fact 2, vertex v is in the pertinent graphs of the children of μ which are ancestors of μ_1 and μ_2. Hence, by Fact 1, vertex v must be in *skeleton*(μ). The second part of the lemma follows from Facts 3 and 4. □

According to Lemma 4.3, the least common ancestor of the allocation nodes of vertex v is called the *proper* allocation node of v. For example, in Figs. 1 and 4, the proper allocation nodes of vertices 1, 6, and 15 are α_1, α_6, and α_5, respectively.

FACT 5. *If $v \neq s, t$, then the proper allocation node of v is either an R-node or an S-node.*

Let v be a vertex of the pertinent graph of a node μ of $\mathcal{T}$. The *representative* of v in *skeleton*(μ) is the vertex or edge x of *skeleton*(μ) defined as follows: if μ is an allocation node of v, then $x = v$; otherwise, x is the edge of *skeleton*(μ) whose expansion graph contains v. For example, in Figs. 1 and 4, edge $(11, 15)$ of *skeleton*(α_5) is the representative of vertices 13, 14, 18, and 19, while vertex 7 in *skeleton*(α_3) is the representative of itself.

FACT 6. *The nodes of $\mathcal{T}$ whose skeleton has a representative for vertex v are the allocation nodes of v (the representative is a vertex) and their ancestors (the representative is an edge).*

LEMMA 4.4. *Given any two distinct vertices v_1 and v_2 of G, there exists a node χ of $\mathcal{T}$ such that v_1 and v_2 have distinct representatives in skeleton(χ). Also, let μ_1 and μ_2 be the proper allocation nodes of v_1 and v_2, respectively, and let μ be the least common ancestor of μ_1 and μ_2.*

1. *If $\mu_1 = \mu_2 = \mu$, then the common allocation nodes of v_1 and v_2 are exactly those with distinct representatives for v_1 and v_2.*

2. *If $\mu_1 \neq \mu$ and $\mu_2 \neq \mu$, then μ is the only node with distinct representatives for v_1 and v_2.*

3. *If μ_1 is an ancestor of μ_2, then the allocation nodes of v_1 on the path from μ_2 to μ_1 are exactly those with distinct representatives for v_1 and v_2 (such nodes form a path in $\mathcal{T}$).*

Proof. By Fact 6, cases 1, 2, and 3 characterize the set of nodes χ of $\mathcal{T}$ such that v_1 and v_2 have distinct representatives in *skeleton*(χ). □

In the example of Figs. 1 and 4, the nodes with distinct representatives for vertices 6 and 17 are α_3 and α_2 (case 3), while for vertices 6 and 13, node α_2 is the only node with distinct representatives (case 2).

A *peripheral* edge (vertex) of a planar st-graph G is such that it appears on the same face of s and t for some embedding of G. A node μ of $\mathcal{T}$ is said to be peripheral if its virtual edge is peripheral in the skeleton of the parent of μ. Note that the children of S- and P-nodes are always peripheral.

In the example of Figs. 1 and 4, the peripheral edges of *skeleton*(α_5) are $(1, 12)$, $(12, 17)$, $(1, 11)$, $(11, 15)$, $(15, 16)$, and $(16, 17)$, while the only nonperipheral vertex of the entire graph is 5. Also, all the R-, P-, and S-nodes except α_9 are peripheral.

FACT 7. *Let e be an edge of the skeleton skeleton(μ) of node μ. If vertex v is peripheral in the expansion graph of e and e is peripheral in skeleton(μ), then v is peripheral in the pertinent graph G_μ of μ.*

The following lemma gives a method for testing whether a vertex is peripheral in the pertinent graph of some node of $\mathcal{T}$.

LEMMA 4.5. *Let ξ be a node of $\mathcal{T}$, and v be a vertex of the pertinent graph G_ξ of ξ. Let μ be the proper allocation node of v.*

1. *If $\mu = \xi$, then v is peripheral in G_ξ if and only if v is peripheral in skeleton(μ).*
2. *If μ is an ancestor of ξ, then v is always peripheral in G_ξ.*
3. *If μ is a descendant of ξ, then let λ be the child of ξ whose subtree contains μ.*

Then vertex v is peripheral in G_ξ if and only if v is peripheral in skeleton(μ) and all the nodes on the path from μ to λ (inclusive) are peripheral.

Proof. Case 1 is proved by observing that substituting the virtual edges of *skeleton*(μ) with their expansion graphs or vice versa does not change the peripheral status of the vertices of *skeleton*(μ). Case 2 follows from Lemma 4.3 since v is a pole of G_ξ. Case 3 follows by inductively applying Fact 7. □

4.2. Test algorithm. In this section, we show how to perform operation *Test*. The algorithm is based on the following theorem, whose intuition is illustrated in Fig. 6.

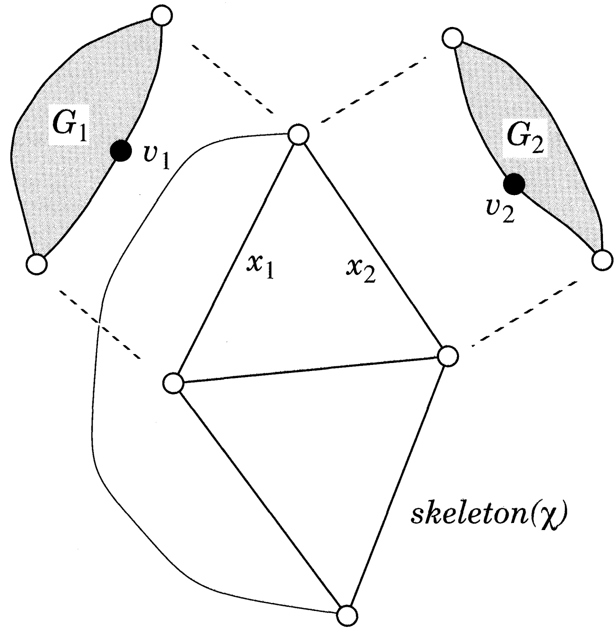


FIG. 6. *Schematic illustration of Theorem* 4.6.

THEOREM 4.6. *Let v_1 and v_2 be vertices of a planar st-graph G with the edge (s,t). There exists an embedding Γ of G such that v_1 and v_2 are on the same face of Γ if and only if there exists a node χ of the decomposition tree $\mathcal{T}$ of G such that:*

1. *v_1 and v_2 have distinct representatives x_1 and x_2 in χ;*
2. *x_1 and x_2 are on the same face of some embedding of skeleton(χ); and*
3. *v_1 and v_2 are peripheral vertices of the expansion graphs G_1 and G_2 of x_1 and x_2, respectively.*

Proof: *If*. From condition 2, let Σ be a planar embedding of *skeleton*(χ) with x_1 and x_2 on the same face. Also, from condition 3, let Γ_1 and Γ_2 be planar embeddings of G_1 and G_2 with vertices v_1 and v_2 on the same face as the poles of G_1 and G_2, respectively. (See Fig. 6.) We replace x_1 and x_2 in Σ with Γ_1 and Γ_2 and perform at most two reverse operations such that v_1 and v_2 will be on the same face. We replace the remaining edges of *skeleton*(χ) with planar embeddings of the corresponding pertinent graphs. This gives a planar embedding of the pertinent graph of χ such that v_1 and v_2 are on the same face. Such an embedding can be easily extended to an embedding of G with the desired property.

Only if. We find node χ using Lemma 4.4. Let μ_1 and μ_2 be the proper allocation nodes of v_1 and v_2, respectively, and let μ be the least common ancestor of μ_1 and μ_2. If one of μ_1 and μ_2 is the ancestor of the other, then we define χ as the lowest node on the path between μ_1 and μ_2 such that the pertinent graph of χ contains both v_1 and v_2. Otherwise, we define $\chi = \mu$. Hence condition 1 is verified.

Consider a planar embedding Γ of G with v_1 and v_2 on the same face. We contract into single edges the pertinent graphs of the children of χ while preserving the embedding. We obtain an embedding of *skeleton*(χ) with x_1 and x_2 on the same face (condition 2).

In order to prove condition 3, assume for contradiction that v_1 is not a peripheral vertex of G_1. We have $\mu_1 \neq \chi$ and μ_1 is an R-node. Let s_1 and t_1 be the poles of *skeleton*(μ_1). By condition 2, x_1 and x_2 are on the same face of *skeleton*(χ), so that there exists a simple undirected path in *skeleton*(χ) between s_1 and t_1 that contains x_2. We replace each edge x of such path with a path π_x between the poles of the pertinent graph of node ν such that x is the virtual edge of ν, where, if x_2 is an edge, path π_{x_2} goes through vertex v_2. This gives a simple undirected path π of G between s_1 and t_1. Hence graph G_1^+ consisting of G_1, edge (v_1, v_2), and path π must be planar. Consider a planar embedding of G_1^+. By removing edge (v_1, v_2) and path π, we obtain a planar embedding of G_1 such that v_1, s_1, and t_1 are on the same face. This contradicts the assumption that v_1 is not a peripheral vertex of G_1. A similar argument can be used to show that v_2 must be a peripheral vertex of G_2. This completes the proof of condition 3. □

In the example of Figs. 1 and 4, vertices 6 and 13 verify the hypothesis of Theorem 4.6, while vertices 5 and 18 do not.

We remark that either a skeleton graph admits a unique embedding (R-node) or any two vertices/edges can be placed on the same face (P-, Q-, and S-nodes). Hence Theorem 4.6 reduces the *Test* operation to a test on a fixed embedding. The algorithm for operation *Test* (Algorithm 1) is based on Theorem 4.6 and its proof.

We give two examples for the algorithm *Test* which refer to Figs. 1 and 4.

Consider operation *Test*$(13, 6)$; namely, $v_1 = 13$ and $v_2 = 6$. We have that $\mu_1 = \alpha_{10}$, $\mu_2 = \alpha_6$, and $\mu = \alpha_2$. Thus we are in case (b) and $\chi = \mu = \alpha_2$, $\lambda_1 = \alpha_5$, $\lambda_2 = \alpha_3$, and $\kappa_1 = \kappa_2 = \alpha_0$. Since χ is a P-node, there exists an embedding of *skeleton*(χ) with the representatives of v_1 and v_2 on the same face. Also, both κ_1 and κ_2 are on the path from χ to the root (actually, they are the root). Therefore, *Test*$(13, 6)$ returns *true.*

Consider operation *Test*$(17, 5)$; namely, $v_1 = 17$ and $v_2 = 5$. We have that $\mu_1 = \alpha_0$, $\mu_2 = \alpha_9$, and $\mu = \mu_1 = \alpha_0$. Thus we are in case (c). Vertex v_2 is peripheral in *skeleton*(μ_2), $\kappa_2 = \alpha_9$, and $\chi = \alpha_6$ and is not an allocation node of v_1. Therefore, *Test*$(17, 5)$ returns *false.* In this example, vertex 5 is not "peripheral enough" with respect to vertex 17.

The correctness of the algorithm follows from Theorem 4.6 and Lemmas 4.4 and 4.5. In case (a), condition 3 of Theorem 4.6 is always trivially verified. Re-

```
ALGORITHM 1. Test($v_1, v_2$)
    1. Find the proper allocation nodes $\mu_1$ of $v_1$ and $\mu_2$ of $v_2$
    2. Find the least common ancestor $\mu$ of $\mu_1$ and $\mu_2$.
    3. case of
       (a) $\mu_1 = \mu_2 = \mu$;
           let $\chi = \mu$;
           if $v_1$ and $v_2$ are on the same face of some embedding of skeleton($\chi$)
               then return true
               else return false.
       (b) $\mu_1 \neq \mu$ and $\mu_2 \neq \mu$;
           let $\chi = \mu$;
           for $i = 1, 2$ do
               Find the representative $x_i$ of $v_i$ in skeleton($\chi$) as follows: de-
               termine the child $\lambda_i$ of $\chi$ on the path from $\mu_i$ to $\chi$, and let $x_i$
               be the virtual edge of $\lambda_i$ in skeleton($\chi$).
               Find the first nonperipheral node $\kappa_i$ on the path from $\mu_i$ to
               the root.
           endfor
           if ($x_1$ and $x_2$ are on the same face of some embedding of
           skeleton($\chi$)) and ($v_1$ and $v_2$ are peripheral vertices of skeleton($\mu_1$)
           and skeleton($\mu_2$), respectively) and ($\kappa_1$ and $\kappa_2$ are either children of
           $\mu$ or on the path from $\mu$ to the root)
               then return true
               else return false.
       (c) $\mu_1 = \mu$ and $\mu_2 \neq \mu$
           if $v_2$ is not a peripheral vertex of skeleton($\mu_2$)
               then return false
           Determine the first nonperipheral node $\kappa_2$ of the path from $\mu_2$ to the
           root.
           if $\kappa_2$ is a child of $\mu$ or $\kappa_2$ is on the path from $\mu_1$ to the root
               then set $\chi = \mu_1$
               else set $\chi$ equal to the parent of $\kappa_2$.
           if $\chi$ is not an allocation node of $v_1$
               then return false
           Find the representative $x_2$ of $v_2$ in skeleton($\chi$).
           if $v_1$ and $x_2$ are on the same face of the embedding of skeleton($\chi$)
               then return true
               else return false.
       (d) $\mu_2 = \mu$ and $\mu_1 \neq \mu$
           (This is analogous to the previous case and therefore omitted.)
    endcase
```

garding condition 2, if it is not verified at node μ, then by Lemma 4.4, μ is the only node that verifies condition 1. In case (b), condition 1 of Theorem 4.6 is verified only for node μ, the least common ancestor of the proper allocation nodes of v_1 and v_2. We set $\chi = \mu$, test condition 2 directly, and verify condition 3 by applying Lemma 4.5.

Case (c) is more complex since more than one node may satisfy condition 1 of Theorem 4.6. By Lemma 4.4, such nodes are the allocation nodes of v_1 on the path from μ_2 to μ_1.

In this case, the specific choice of node χ made in the proof of Theorem 4.6 (i.e., the lowest node on the path between μ_1 and μ_2 such that the pertinent graph of χ contains both v_1 and v_2), which satisfies condition 1, appears difficult to compute. Thus we use a slightly different approach, where node χ satisfies condition 3. First, we choose node χ as the highest node where condition 3 is satisfied by applying Lemma 4.5. If χ is not an allocation node of v_1, then condition 1 is not verified at χ; moreover, since condition 1 can be satisfied only at ancestors of χ and condition 3 can be satisfied only at or below χ, there is no node for which both conditions 1 and 3 can be satisfied. Otherwise (χ is an allocation node of v_1), condition 1 is satisfied and we check condition 2 directly. If condition 2 is not verified, then v_1 is not an endpoint of the representative edge x_2 of v_2 in *skeleton*(χ). Hence v_1 is not a pole of the expansion graph of x_2, so that no descendant of χ is an allocation node of v_1. This implies that condition 1 cannot be verified at any descendant of χ.

4.3. Static data structure and time complexity. The following data structure can be used to efficiently perform the *Test* operation in a static environment. We store with each vertex a pointer to its proper allocation node in $\mathcal{T}$. Hence step 1 takes $O(1)$ time. We equip tree $\mathcal{T}$ with a data structure which uses linear space and supports least common ancestor queries in constant time [29, 50]. Hence step 2 takes $O(1)$ time.

Concerning step 3, we set up the following data structures. Each node of $\mathcal{T}$ has a pointer to the corresponding virtual edge in the skeleton of its parent. We mark all the peripheral nodes and the peripheral vertices and edges of each skeleton. Also, each node ζ has a pointer to the first nonperipheral node $\hat{\zeta}$ in the path from ζ to the root (the root node points to itself). Finally, we equip the skeleton of each R-node with the data structure for planar embedding tests described in [53]. This allows us to test whether two vertices/edges are on the same face of the planar embedding of the skeleton in $O(\log n)$ time.

By Lemma 4.1, tree $\mathcal{T}$ uses $O(n)$ space. All the remaining data structures use $O(n)$ space. The decomposition tree $\mathcal{T}$ can be constructed in $O(n)$ time using a variation of the algorithm of [30] for finding the triconnected components of a graph. The planar embeddings of the skeletons of $\mathcal{T}$ and their peripheral vertices and edges can be computed in $O(n)$ time using the planarity-testing algorithm of [30]. We conclude the following.

THEOREM 4.7. *Let G be a biconnected planar graph with n vertices. There exists an $O(n)$-space data structure that supports operation Test(u,v) on G in time $O(\log n)$ and can be constructed in $O(n)$ preprocessing time.*

Proof. Orient G into a planar st-graph, where s and t are adjacent vertices, and then use algorithm *Test* with the data structure described above. □

Note that the $O(\log n)$ bound on the query time depends only on the performance of the data structure of [53] for testing whether two vertices/edges are on the same face of a planar embedding. It can be shown that, applying perfect hashing [59], the query time of [53] can be reduced to $O(1)$ at the expense of using a complicated $O(n)$-space data structure with $O(n^2)$ preprocessing time. We have the following corollary.

COROLLARY 4.8. *Let G be a biconnected planar graph with n vertices. There exists an $O(n)$-space data structure that supports operation Test(u,v) on G in time $O(1)$ and can be constructed in $O(n^2)$ preprocessing time.*

5. Updates. In this section, we show how to perform operations *InsertEdge* and *InsertVertex* on a biconnected planar graph G. As shown in the following theorem, the above repertoire of update operations is complete for biconnected planar graphs.

THEOREM 5.1. *A biconnected planar graph G with $n \geq 3$ vertices and m edges can be assembled starting from the triangle graph (a cycle of three vertices) by means of $m - 3$ InsertEdge and InsertVertex operations such that each intermediate graph is planar and biconnected. Also, such a sequence of operations can be determined in $O(n)$ time.*

Proof. We compute an *open-ear decomposition* $D = (P_0, P_1, \ldots, P_r)$ of G, which is a partition of the edges of G into an ordered collection of edge-disjoint simple paths $P_0, P_1, \ldots, P_r$, called *ears*, such that

- P_0 is a simple cycle;
- the two endpoints of ear P_i, for $i \geq 1$, are distinct and contained in some P_j, $j < i$; and
- none of the internal vertices of P_i are contained in any P_j, $j < i$.

A graph G has an open-ear decomposition if and only if it is biconnected; moreover, all intermediate graphs $D_i = P_0 + P_1 + \cdots + P_i$ of an open-ear decomposition of a biconnected graph are biconnected [61]. Further, if G is planar, then each D_i is planar since it is a subgraph of G. An ear decomposition can be computed in $O(n)$ time using the st-numbering technique [18]. We show how to use D to determine the assembly sequence of G. Starting from the initial triangle graph, we construct cycle P_0 by means of a sequence of *InsertVertex* operations. Next, we add the remaining ears $P_1, \ldots, P_r$, each by means of one *InsertEdge* operation followed by zero or more *InsertVertex* operations. Since we start with the triangle graph having three vertices and since each operation adds one edge, the total number of operations is $m - 3$. To avoid forming intermediate graphs with multiple edges, we modify the ears as follows. For each edge $e = (u, v)$, let P_{i_0} be the ear containing e, with $P_{i_0} = P'eP''$. If there are ears $P_{i_1}, \ldots, P_{i_k}$ with endpoints u and v and $i_0 < i_1 < \cdots < i_k$, we replace P_{i_0} with $P'P_{i_k}P''$ and P_{i_k} with e. Note that each intermediate graph generated is planar since it is homeomorphic to a subgraph of G. The above modification of the ears can be computed in $O(n)$ time by radix-sorting the edges and ears on their endpoints. □

In our dynamic environment, we maintain a planar st-orientation of a biconnected planar graph G such that s and t are adjacent vertices as follows.

In operation *InsertVertex*(e, v, e_1, e_2), if e goes from s to t, then we orient edge e_1 from s to v and edge e_2 from t to v. Vertex v is the new sink of the orientation. Otherwise, we orient e_1 and e_2 in the same way as e.

In operation *InsertEdge*(e, v_1, v_2), we orient e from v_1 to v_2 if v_2 is reachable from v_1 in the planar st-orientation and from v_2 to v_1 if v_1 is reachable from v_2. If neither vertex is reachable from the other, both orientations of e are possible. To test the condition on reachability, we use the following theorem.

THEOREM 5.2. *Let v_1 and v_2 be vertices of a planar st-graph G with the edge (s, t) and such that there exists an embedding Γ of G such that v_1 and v_2 are on the same face of Γ. Let χ be a node of the decomposition tree $\mathcal{T}$ of G such that*

1. *v_1 and v_2 have distinct representatives x_1 and x_2 in χ;*
2. *x_1 and x_2 are on the same face f of some embedding of skeleton(χ); and*
3. *v_1 and v_2 are peripheral vertices of the expansion graphs G_1 and G_2 of x_1 and x_2, respectively.*

Then there exists a directed path in G from v_1 to v_2 if and only if there exists a directed path in the boundary of face f from x_1 to x_2.

Proof. Note that the existence of node χ is guaranteed by Theorem 4.6. By the definition of a pertinent graph, there exists a directed path in G from v_1 to v_2 if and only if there exists a directed path in *skeleton*(χ) from x_1 to x_2. By the reachability properties of planar st-graphs given in [54], we have that there exists a directed path in $skeleton(\chi)$ from x_1 to x_2 if and only if there exists a directed path in the boundary of face f from x_1 to x_2. □

By property 2 of Lemma 3.1, face f consists of two directed paths with a common origin and destination. Hence the time for testing reachability in f is dominated by the complexity of determining if two objects of f (each a vertex or an edge) are on the same path and, if so, which object precedes the other. In the rest of this section, we assume that G is a planar st-graph that contains the edge (s, t).

The algorithm for operation *InsertVertex* is as follows. Let ρ be the Q-node storing edge e and let π be the parent of ρ. If π is a P-node or an R-node, we replace ρ with a subtree consisting of an S-node and two child Q-nodes. If π is an S-node, we remove ρ and add two new child Q-nodes to π.

In the rest of this section, we present the algorithm for operation *InsertEdge*(e, v_1, v_2). The algorithm makes use of several types of transformations that modify the decomposition tree. If a transformation produces a node with exactly one child, such node is absorbed into its parent. We assume that operation *Test*(v_1, v_2) has been already performed and has returned *true*.

The algorithm *InsertEdge* and its subroutines are shown as Algorithm 2 and Procedures 1–6.

```
ALGORITHM 2. InsertEdge($e, v_1, v_2$)
Find the proper allocation nodes $\mu_1$ of $v_1$ and $\mu_2$ of $v_2$ and their least common
ancestor $\mu$.
case of
    1. $\mu_1 = \mu_2 = \mu$;
            let $\chi = \mu$;
            { Since $G$ already contains the edge $(s,t)$, by Fact 5, node $\chi$ is
            either an S-node or an R-node. }
            FinalTransformation1 ($\chi$)
    2. $\mu_1 \neq \mu$ and $\mu_2 \neq \mu$;
       let $\chi = \mu$;
       for $i = 1, 2$ do
            PathCondensation ($\mu_i, \lambda_i$)
       endfor
       FinalTransformation2 ($\chi, \lambda_1, \lambda_2$).
    3. $\mu_1 = \mu$ and $\mu_2 \neq \mu$;
       Determine the lowest node $\chi$ on the path from $\mu_2$ to $\mu$ such that skeleton($\mu$)
       contains $v_1$.
       if $\chi = \mu_2$
       then FinalTransformation1 ($\chi$)
       else
            PathCondensation ($\mu_2, \lambda_2$);
            FinalTransformation3 ($\chi, \lambda_2$).
    4. $\mu_2 = \mu$ and $\mu_1 \neq \mu$
       (This is analogous to the previous case and therefore omitted.)
endcase
```

PROCEDURE 1. *FinalTransformation*1 (χ)

1. χ *is an R-node.* We have two subcases.
   Graph *skeleton*(χ) does not contain an edge between v_1 and v_2.
   Edge e is inserted in *skeleton*(χ) and we add to the children of χ a new Q-node associated with e.
   Graph *skeleton*(χ) contains an edge between v_1 and v_2.
   The edge (v_1, v_2) of *skeleton*(χ) is the virtual-edge of a child ν of χ. If ν is a P-node, we add to the children of ν a new Q-node associated with e and we insert another edge from v_1 to v_2 in *skeleton*(ν). Else, we replace ν with a new P-node with children ν and a Q-node storing edge e.
2. χ *is an S-node.* We perform at node χ the transformation illustrated in Fig. 7. The sequence of children of χ is partitioned into subsequences α, β, and γ, where β consists of the children of χ associated with the edges of *skeleton*(χ) between vertices v_1 and v_2. We remove the nodes of β from the children of χ and replace them with a new P-node whose children are a Q-node associated with edge e and an S-node whose children are the nodes of β. Graph *skeleton*(χ) is updated by replacing the chain between v_1 and v_2 with a single edge. The skeleton of the new P-node consists of two multiple edges from v_1 to v_2. The skeleton of the new S-node consists of a chain of $|\beta|$ edges from v_1 to v_2. Note that if $|\beta| = 1$, the new S-node is absorbed into its parent, as mentioned above.

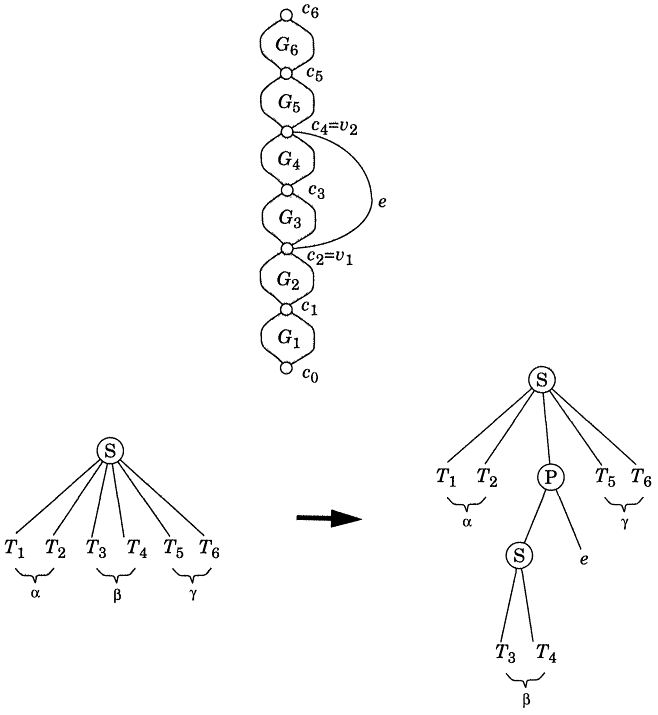


FIG. 7. *The procedure FinalTransformation*1 *when* χ *is an S-node.*

```
PROCEDURE 2. PathCondensation ($\mu_i$, $\lambda_i$)

        InitialTransformation ($\mu_i$)
        Determine the child $\lambda_i$ of $\chi$ on the path from $\mu_i$ to $\chi$.
        set $\rho = \mu_i$;
        while $\rho \neq \lambda_i$ do
            set $\pi$ equal to the parent of $\rho$;
            ElementaryTransformation ($\rho, \pi; \pi'$);
            set $\rho = \pi'$;
        endwhile
```

```
PROCEDURE 3. InitialTransformation ($\mu_i$)

        If $\mu_i$ is an S-node, expand $\mu_i$ into a structure consisting of an R-
        node $\nu$ and two S-nodes $\nu'$ and $\nu''$, such that (see Fig. 8)
          • $\nu$ has children $\nu'$ and $\nu''$ and has the same parent as $\mu_i$;
          • graph skeleton($\nu$) consists of edges $(s_{\mu_i}, v_i)$, $(v_i, t_{\mu_i})$, and
            $(s_{\mu_i}, t_{\mu_i})$;
          • graphs skeleton($\nu'$) and skeleton($\nu''$) consist of the subchains
            of skeleton($\mu_i$) from $s_{\mu_i}$ to $v_i$ and from $v_i$ to $t_{\mu_i}$, respectively.
        Rename $\mu_i = \nu$.
```

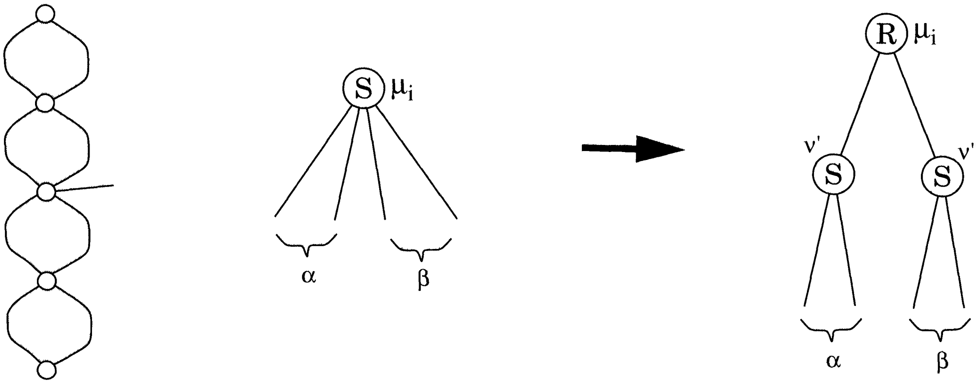


FIG. 8. *Expansion of an S-node in InitialTransformation.*

PROCEDURE 4. *ElementaryTransformation* $(\rho, \pi; \pi')$

Let X be the type of node ρ and let π be the parent of ρ. Perform the RX-transformation described below and shown in Fig. 9.

RR-transformation: Contract nodes ρ and π into a new node π'. Graph *skeleton*(π') is obtained from *skeleton*(π) by replacing the virtual edge of ρ with *skeleton*(ρ) minus the edge (s_ρ, t_ρ). (See Fig. 9(a))

RP-transformation: Rename ρ and π into π' and ρ', respectively. Set the parent of ρ' equal to π'. Set the parent of π' equal to the former parent of π. Graph *skeleton*(ρ') is equal to *skeleton*(π) minus one of the edges (s_ρ, t_ρ) (the former virtual edge of ρ). Graph *skeleton*(π') is equal to *skeleton*(ρ) plus a virtual edge $(s_{\rho'}, t_{\rho'})$. (See Fig. 9(b))

RS-transformation: Split node π into nodes ρ' and ρ'' such that *skeleton*(ρ') and *skeleton*(ρ'') are the subchains of *skeleton*(π) from s_π to s_ρ and from t_ρ to t_π, respectively. Rename ρ into π'. Set the parents of ρ' and ρ'' equal to π'. Set the parent of π' equal to the former parent of π. Graph *skeleton*(π') consists of *skeleton*(ρ) minus edge (s_ρ, t_ρ) plus edges (s_π, s_ρ), (t_ρ, t_π), and (s_π, t_π). (See Fig. 9(c))

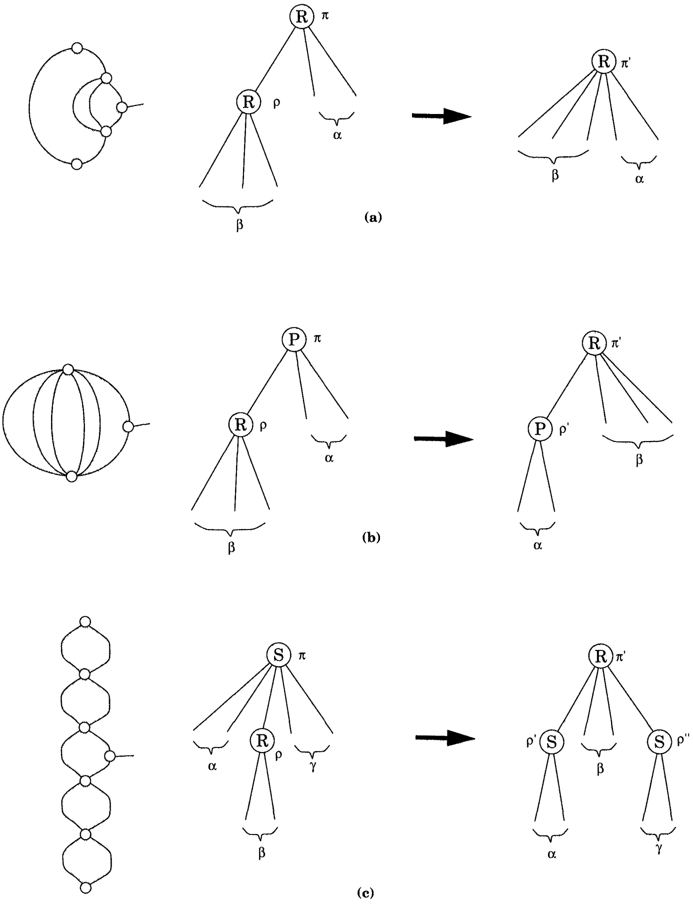


FIG. 9. *Elementary transformations:* (a) *RR*; (b) *RP*; (c) *RS*.

PROCEDURE 5. *FinalTransformation2* $(\chi, \lambda_1, \lambda_2)$

Let X be the type of node χ. Perform the X-transformation described below and shown in Fig. 10. Note that λ_1 and λ_2 are R-nodes.

R-transformation: Contract nodes χ, λ_1 and λ_2 into a new R-node χ'. Graph *skeleton*(χ') is obtained from *skeleton*(χ) by replacing the virtual edges of λ_1 and λ_2 with their skeletons (minus the edge between their poles) and by adding the edge (v_1, v_2).

P-transformation: Contract nodes λ_1 and λ_2 into a new R-node λ. Graph *skeleton*(λ) is obtained by the union of *skeleton*(λ_1), *skeleton*(λ_2), and the edge (v_1, v_2).

S-transformation: Partition the sequence of children of χ into subsequences α, λ_1, β, λ_2, and γ in this order from left to right. We remove the nodes of β from the children of χ and replace them with a new R-node ν. Also, we create a new S-node λ with parent ν and whose children are the nodes of β. Graph *skeleton*(ν) is obtained from *skeleton*(λ_1) and *skeleton*(λ_2) by adding the edges $(s_{\lambda_1}, t_{\lambda_2})$, $(t_{\lambda_1}, s_{\lambda_2})$, and (v_1, v_2).

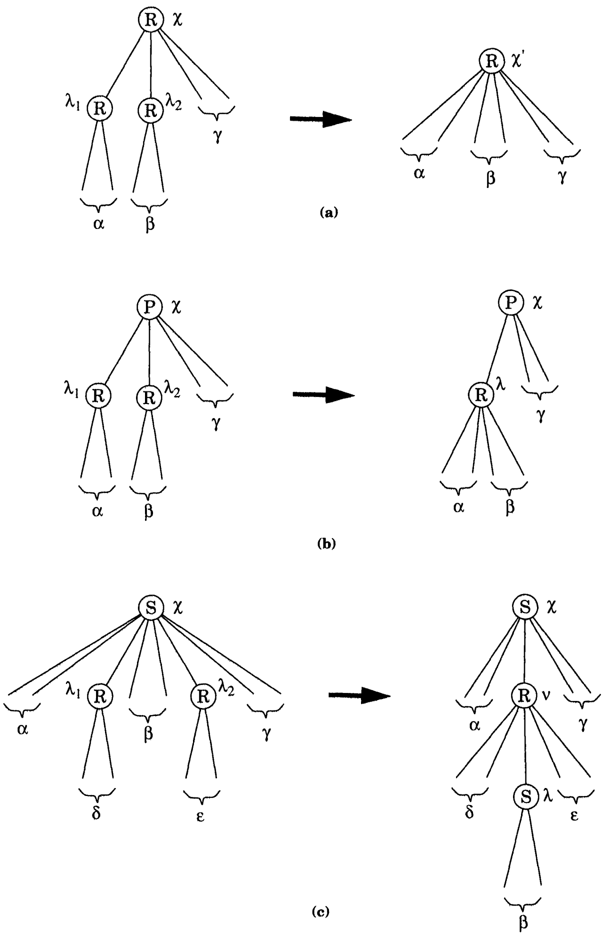


FIG. 10. *The procedure FinalTransformation2:* (a) R; (b) P; (c) S.

PROCEDURE 6. *FinalTransformation3* (χ, λ_2)

Let X be the type of node χ. Perform the X-transformation described below. Note that λ_2 is an R-node.

R-transformation: Contract nodes χ and λ_2 into a new R-node χ'. Graph *skeleton*(χ') is obtained from *skeleton*(χ) by replacing the virtual edge of λ_2 with *skeleton*(λ_2) (minus the edge between the poles) and by adding the edge (v_1, v_2).

S-transformation: Partition the sequence of children of χ into subsequences α, β, λ_2, and γ in this order from left to right, where v_1 is the common pole of the pertinent graphs of the last node of α and the first node of β. We remove the nodes of β from the children of χ and replace them with a new R-node ν. Also, we create a new S-node λ with parent ν and whose children are the nodes of β. Graph *skeleton*(ν) is obtained from *skeleton*(λ_2) by adding the edges (v_1, s_{λ_2}), (v_1, t_{λ_2}), and (v_1, v_2).

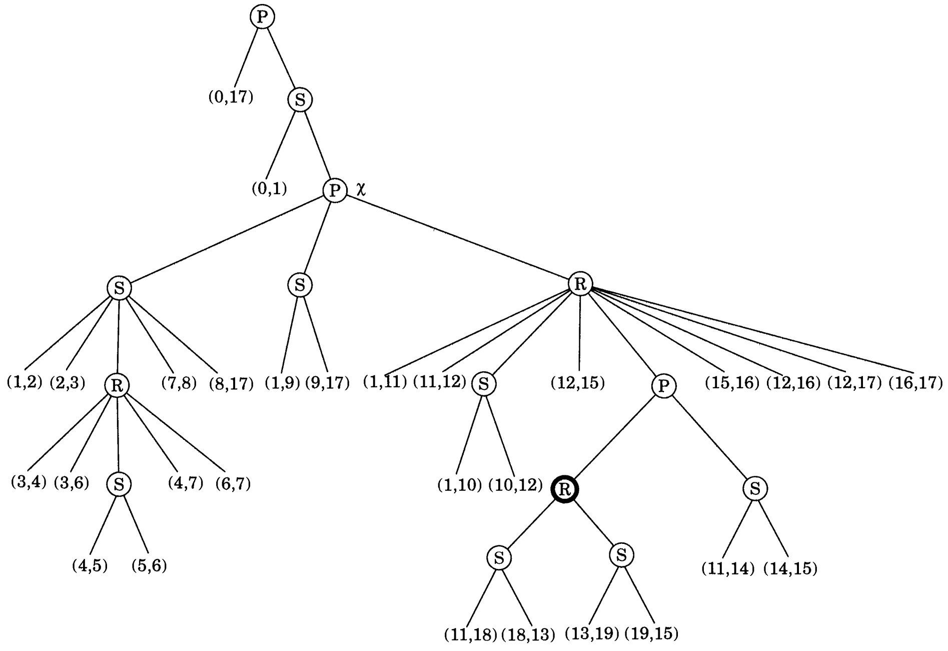


FIG. 11. *Initial S-transformation in operation InsertEdge*$(e, 13, 6)$.

With reference to Figs. 1 and 4, we show how *InsertEdge*$(e, 13, 6)$ is performed. We are in case 2. The initial transformation at μ_1 is shown in Fig. 11. Elementary transformations RP, RR, and RS are shown in Figs. 12–14. The final decomposition tree (after *FinalTransformation2*) is shown in Fig. 15.

We now argue about the correctness of the algorithm *InsertEdge*. We discuss case 2 since it is the most general. Similar considerations hold for cases 1, 3, and 4.

First, we observe that after the insertion of edge (v_1, v_2), the poles of χ remain a separation pair of G. Hence only the subtree of $\mathcal{T}$ rooted at χ is affected by the insertion. Namely, we show that the algorithm *InsertEdge* correctly computes the decomposition tree of the pertinent graph of χ plus the edge (v_1, v_2).

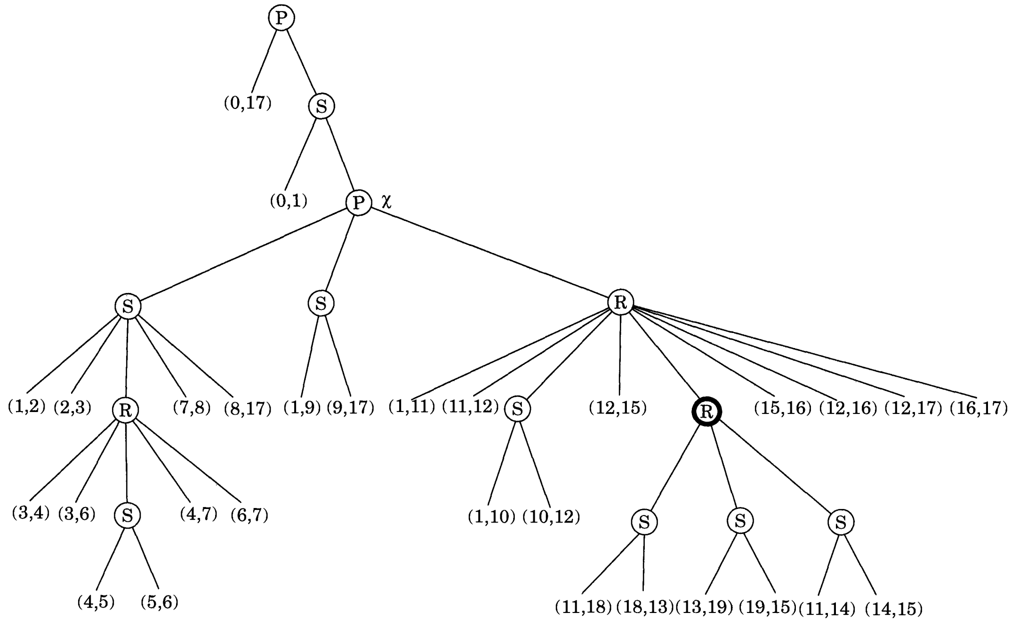


FIG. 12. *Elementary RP-transformation in operation InsertEdge*$(e, 13, 6)$.

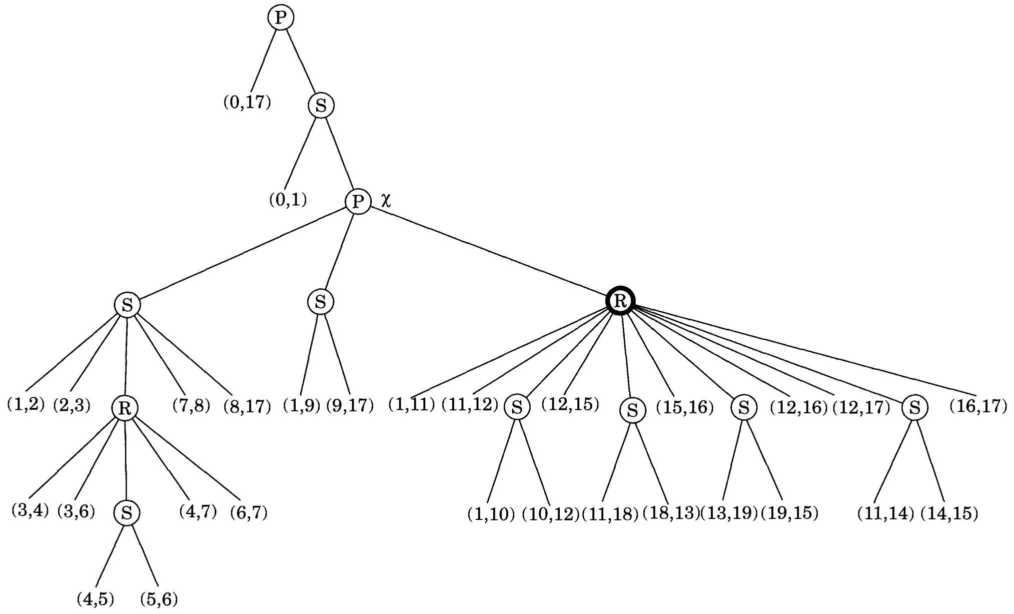


FIG. 13. *Elementary RR-transformation in operation InsertEdge*$(e, 13, 6)$.

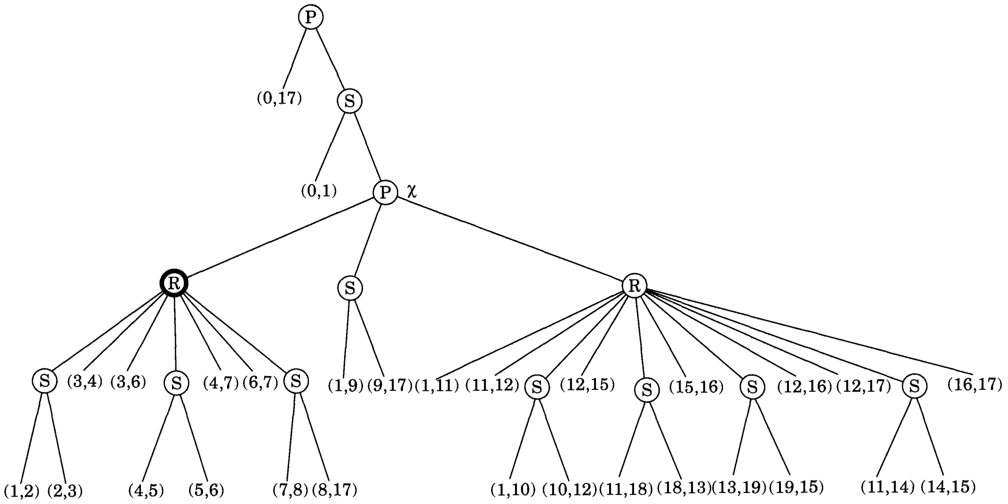


FIG. 14. *Elementary RS-transformation in operation InsertEdge*$(e, 13, 6)$.

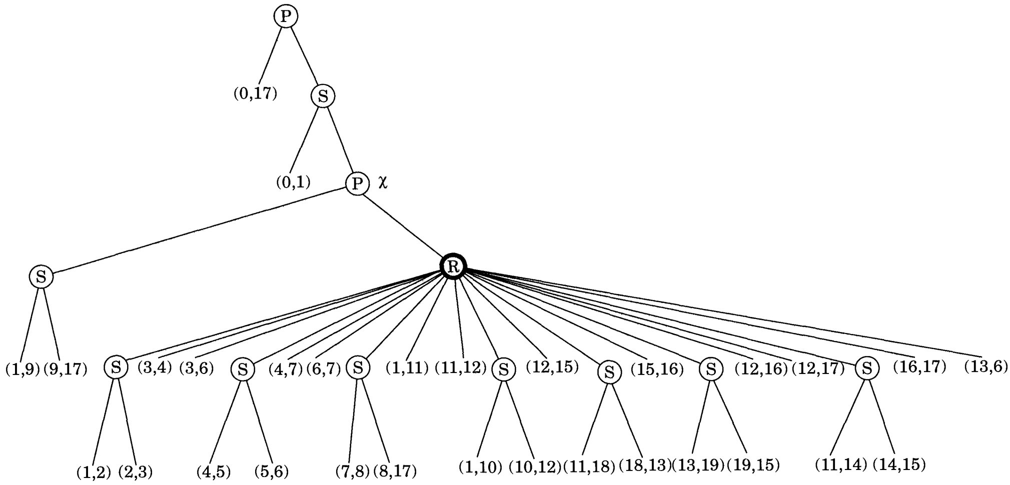


FIG. 15. *Final P-transformation in operation InsertEdge*$(e, 13, 6)$.

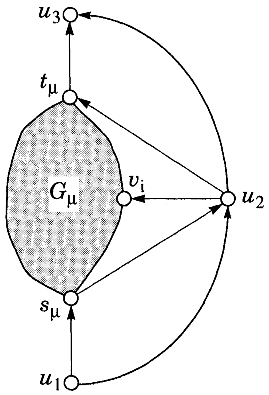


FIG. 16. *Graph Θ^i_μ used to show the correctness of the algorithm InsertEdge.*

Let μ be a node of $\mathcal{T}$ whose pertinent graph G_μ contains vertex v_i. We denote by Θ^i_μ the graph obtained from G_μ by adding three new vertices u_1, u_2, and u_3 and the edges (u_1, s_μ), (u_1, u_2), (s_μ, u_2), (u_2, v_i), (u_2, t_μ), (u_2, u_3), and (t_μ, u_3), as shown in Fig. 16. Intuitively, the "gadget" added to G_μ forces v_i to appear on the external face. The formal proof consists of showing that

1. each transformation (initial or elementary) at a node π on the path from μ_i to λ_i produces the decomposition tree of Θ^i_π, $i = 1, 2$, except for the Q-nodes associated with the extra edges added to G_μ and their virtual edges in the skeletons;
2. the final transformation at node χ produces the decomposition tree of the pertinent graph of χ plus the edge (v_1, v_2).

The first property can be proved by induction. The base case is the initial transformation at μ_i. The inductive steps correspond to the elementary transformations. The second property can be proved by a simple case analysis. Note that the graph Θ^i_π is planar since by Theorem 4.6 vertex v_i is peripheral in the pertinent graph of π. We omit the details of the correctness proof, which are tedious but straightforward.

6. Dynamic data structure. All the information needed to perform the *Test* algorithm must be updated by the *InsertEdge* and *InsertVertex* algorithms. We describe a data structure that represents the decomposition tree $\mathcal{T}$, the skeletons (with their embeddings) of the nodes of $\mathcal{T}$, and the maximal paths of peripheral nodes in $\mathcal{T}$. The interface of the data structure consists of records for the vertices and edges of the graph G.

6.1. Requirements. In this section, we discuss the primitive operations that need to be supported by the dynamic data structure. The data structure for the decomposition tree $\mathcal{T}$ should support finding the parent of a node and the least common ancestor of two nodes. Also, it should support the initial, elementary, and final transformations. Each such transformation is executed by means of a constant number of link/cut and expand/contract operations.

Concerning skeletons and their embeddings, we need to support a repertoire of access, query, and update operations. The access operations are as follows:

1. find the proper allocation node of a vertex;
2. find the poles of the skeleton of a node.

The query operations are as follows:

1. determine if a vertex is peripheral with respect to the skeleton of an R-node;

2. determine if two objects (each a vertex or an edge) are on the same face of the skeleton of an R-node.

3. determine if two objects on the same face f of a skeleton are also on the same directed path forming the boundary of f and, if so, which object precedes the other.

The nontrivial update operations are as follows:

1. add vertices and edges to skeletons;
2. replace an edge of a skeleton with another skeleton;
3. split the skeleton of an S-node (by removing an edge).

Finally, we need to maintain the set of peripheral nodes of $\mathcal{T}$ so that we can efficiently determine the first nonperipheral node κ on the path from a node μ to the root of $\mathcal{T}$.

6.2. Maintaining planar embeddings. Our technique for maintaining the planar embedding of the skeletons extends that of [53], where the latter two update operations are not supported.

We recall from property 2 of Lemma 3.1 that the boundary of each face of the embedding of a planar *st*-graph G consists of two directed paths with common origin and destination. Also, by property 1 of Lemma 3.1, each vertex of G distinct from the poles s and t is an internal node of exactly two faces.

FACT 8 (see [53]). *Let Γ be a planar embedding of a planar st-graph G. Objects x_1 and x_2 of G, each a vertex or an edge, are on the same face f of Γ if and only if one of the following conditions is verified:*

1. *each of x_1 and x_2 is an edge or an internal vertex of f;*
2. *one of x_1 and x_2 is an edge or an internal vertex and the other is an extreme vertex of f;*
3. *both x_1 and x_2 are extreme vertices of f.*

An *extreme pair* is a pair of vertices that are the extreme vertices of some face f of Γ. For example, $(12, 17)$ and $(11, 15)$ are extreme pairs of the graph of Fig. 1.

By Lemma 3.1, every object is internal in exactly two faces, and every face has exactly two extreme objects (always vertices). Hence conditions 1 and 2 can be tested in constant time after having determined the four faces where x_1 and x_2 are internal and these faces' extreme vertices. Condition 3, on the other hand, is tested by searching for (v_1, v_2) in the set of extreme pairs. The data structure of [53] maintains the set of extreme pairs in a dynamic dictionary and the set of internal vertices of each face in two concatenable queues (associated with the two directed paths forming its boundary).

We now show how to modify the data structure of [53] to support our extended set of operations.

LEMMA 6.1. *The set of extreme pairs is an invariant of a planar st-graph with respect to all its planar embeddings.*

Proof. By Lemma 4.2 we can construct any embedding by means of reverse and swap operations on a given embedding. We show that the set of extreme pairs of an embedding stays unchanged after a reverse or a swap. Consider a reverse operation on a split component C with poles s' and t' (see Fig. 17). The boundaries of the faces internal to C are modified by exchanging their left and right chains, so that their extreme pairs remain the same. Let γ' and γ'' be the left and right chains forming the external boundary of C, where each such chain does not contain s' or t'. The faces f and g on the left and right of C contain γ' and γ'' as subchains of their right and left chains, respectively. After the reverse operation, these faces are modified by replacing γ' with γ'' in the right chain of f and γ'' with γ' in the left chain of g. Hence the extreme pairs of f and g stay unchanged. Finally, the remaining faces of G are not affected by the reverse operation. Similar considerations show that extreme pairs stay unchanged after a swap operation. □

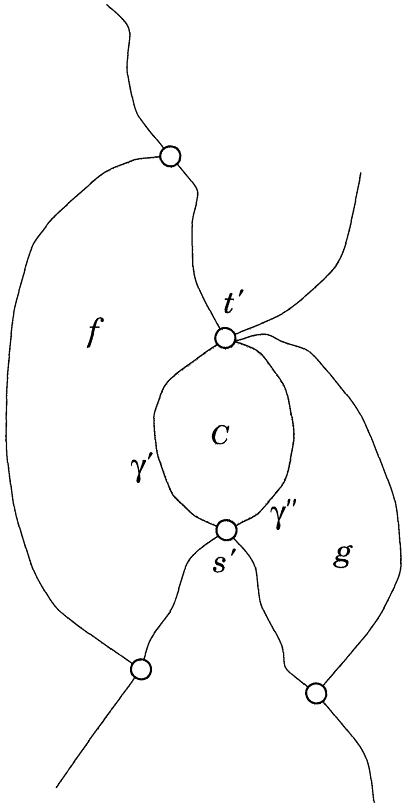


FIG. 17. *Example for the proof of Lemma* 6.1.

According to Lemma 6.1, we denote by $\mathcal{E}$ the set of extreme pairs of G.

LEMMA 6.2. *After performing operation InsertEdge*(e, v_1, v_2)*, the set* $\mathcal{E}$ *is updated by means of at most one deletion and two insertions. Also, after performing operation InsertVertex*(v, e, e_1, e_2)*, the set* $\mathcal{E}$ *stays unchanged.*

Proof. By Lemma 6.1, the update of the set $\mathcal{E}$ is the same in any embedding. Hence we consider adding the edge (v_1, v_2) to an an embedding where v_1 and v_2 are on the same face f. By Lemma 3.1, the boundary face f consists of two directed paths. Let (l, h) be the extreme pair of f. If v_1 and v_2 are on the same directed path of f, then we add to $\mathcal{E}$ the pair (v_1, v_2), unless it is already in $\mathcal{E}$ (when $v_1 = l$ and $v_2 = h$). Otherwise, we remove from $\mathcal{E}$ the pair (l, h) and add to $\mathcal{E}$ the pairs (v_1, h) and (l, v_2). □

The poles of each skeleton and, for R-nodes, the edge between them are directly stored at each node, so that they can be determined in $O(1)$ time. Their update takes $O(1)$ time in each transformation.

The record of each face f of *skeleton*(μ) (except the two faces containing the edge (s_μ, t_μ)) has a bidirectional pointer to the element of $\mathcal{E}$ associated with the extreme pair of f.

6.3. Data structure. The data structure consists of a *main component* and of an *auxiliary component*. The main component is a tree $\mathcal{T}^*$ that represents both the decomposition tree $\mathcal{T}$ and the skeletons of the nodes of $\mathcal{T}$. The auxiliary component is a dictionary (e.g., a balanced search tree) that stores the set $\mathcal{E}$, so that searches and updates in $\mathcal{E}$ take $O(\log n)$ time. The updates to be performed in $\mathcal{E}$ are determined in the final transformations.

The main component $\mathcal{T}^*$ is an *edge-ordered* dynamic tree [15], a variation of the dynamic tree of Sleator and Tarjan [52]. It is a rooted ordered tree with nodes of various types that supports each of the following *primitive tree operations* in logarithmic time:

- Find the parent of a node.
- Find the least common ancestor of two nodes.
- Given two sibling nodes, determine which one precedes the other in the ordered sequence of the children of their parent.
- Given a node ν, find the first node of a given type on the path from ν to the root.
- Link two trees by making the root of one tree a child of a node of the other tree.
- Cut a tree into two trees by removing a tree-edge.
- Expand a node ν into two nodes ν_1 and ν_2 linked by a new tree-edge such that the expansion preserves the ordering of the children. In other words, if $\alpha\beta\gamma$ is the sequence of children of ν, then $\alpha\nu_2\gamma$ is the sequence of children of ν_1 and β is the sequence of children of ν_2.
- Contract a tree-edge (ν_1, ν_2) and merge nodes ν_1 and ν_2 into a new node ν such that the contraction preserves the ordering of the children. In other words, if $\alpha\nu_2\gamma$ is the sequence of children of ν_1 and β is the sequence of children of ν_2, then $\alpha\beta\gamma$ is the sequence of children of ν.

The main component $\mathcal{T}^*$ is obtained from the decomposition tree $\mathcal{T}$ by expanding each node μ of $\mathcal{T}$ into a tree rooted at μ, called a *skeleton tree*, which describes the embedding of *skeleton*(μ), as follows (see Fig. 18):

1. First, we make children of μ a set of *f-nodes* representing the faces of *skeleton*(μ) (their order is irrelevant). The f-node associated with a face f is also said to be a *p-node* ("peripheral" node) if f contains an edge (s_μ, t_μ), and otherwise it is said to be a *b-node* ("blocking" node). Note that if μ is a P-node or an S-node, then all the children of μ are p-nodes. Also, if μ is an S-node or an R-node, it has two child p-nodes.
2. Next, we attach to each f-node two subtrees, called *boundary trees*, that represent the two directed paths forming the boundary of the face (see Lemma 3.1), excluding the extreme vertices. Each boundary tree is a two-level tree whose leaves are an alternating sequence of *e-nodes* and *v-nodes* representing edges and vertices of the path, respectively, and are ordered according to the direction of the path.
3. Finally, for each former child ν of μ, we make ν child of one of the two e-nodes of the skeleton tree of μ associated with the virtual edge of ν in *skeleton*(μ). If the closest f-node ancestor of one of such e-nodes is a p-node, then we make μ a child of that e-node.

The data structure is completed by the following additional pointers. Each R-, P-, and S-node stores pointers to the poles of its skeleton, so that they can be determined in $O(1)$ time. Their update takes $O(1)$ time in each transformation. Each f-node has

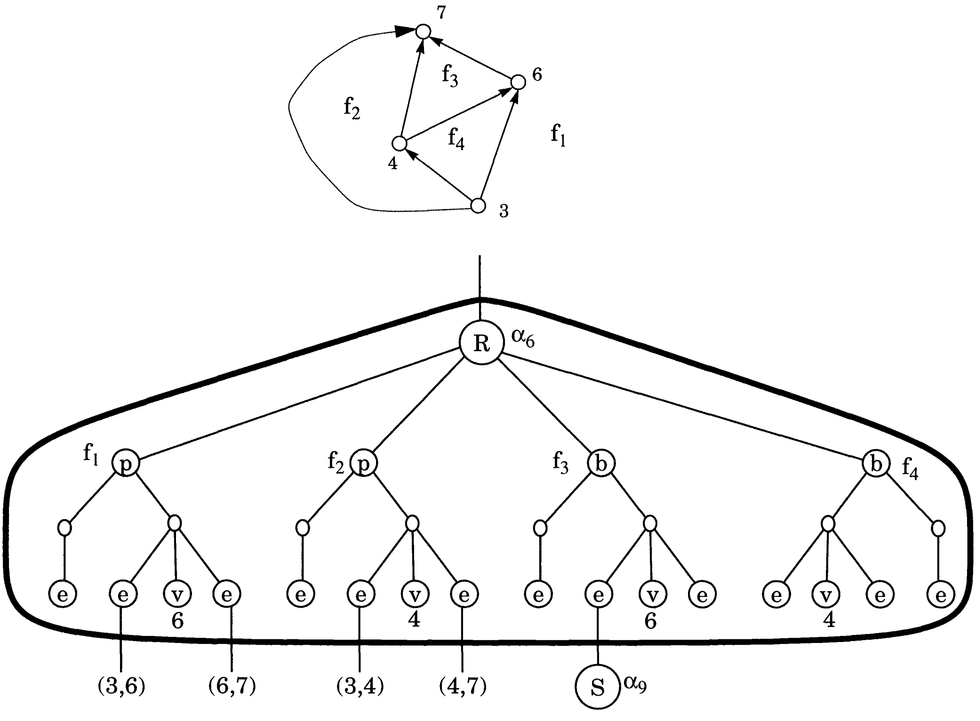


FIG. 18. *Portion of tree $\mathcal{T}^*$ representing the skeleton tree for node α_6 of the decomposition tree $\mathcal{T}$ shown in Fig.* 4.

a bidirectional pointer to the element of $\mathcal{E}$ storing the extreme pair of its associated face. Finally, we establish a pointer from each edge to its Q-node and two pointers from each vertex v to its two representative v-nodes in the boundary trees of the skeleton tree of *skeleton*(μ), where μ is the proper allocation node of v. Note that tree $\mathcal{T}$ can be obtained from tree $\mathcal{T}^*$ by contracting each skeleton tree into its root. We call the S-, P-, Q-, and R-nodes of $\mathcal{T}^*$ *primary nodes.*

6.4. Complexity analysis. In this section, we analyze the performance of the data structure described in §6.3.

LEMMA 6.3. *The above data structure for on-line planarity testing uses $O(n)$ space.*

Proof. The data structure uses space proportional to the total size of the skeletons stored at the nodes of $\mathcal{T}^*$. Thus, by Lemma 4.1, the space requirement is $O(n)$. □

We now consider operation *Test*.

LEMMA 6.4. *The above data structure supports operation Test in time $O(\log n)$.*

Proof. To find the proper allocation node of a vertex v, we access one of the two v-nodes η of v and find the closest primary node ancestor of η. This takes time $O(\log n)$.

To determine whether two objects x_1 and x_2 (each a vertex or an edge) are on the same face of a skeleton, we use Fact 8. Conditions 1 and 2 of Fact 8 are checked in $O(\log n)$ time by finding the closest f-node ancestors of the e-nodes of x_1 and x_2. Namely, x_1 and x_2 are both on face f if the f-node of f is the closest f-node ancestor for both an e-node of x_1 and an e-node of x_2. Condition 3 is verified by searching for the pair (x_1, x_2) in $\mathcal{E}$, again in $O(\log n)$ time.

To determine whether a vertex v is peripheral in the skeleton of its proper allocation node μ, we find the closest f-node ancestors of the two v-nodes of v and test if at least one of them is a p-node. This takes time $O(\log n)$.

To determine the first nonperipheral primary node κ on the path from a node μ to the root of $\mathcal{T}^*$, we find the closest b-node ζ ancestor of μ and then the closest primary node ancestor of ζ. This takes time $O(\log n)$. □

Operation *InsertVertex* is very simple to analyze. It takes worst-case time $O(\log n)$. In the rest of this section, we discuss the time complexity of operation *InsertEdge*.

In operation *InsertEdge*, we need to restructure the tree $\mathcal{T}^*$ and the dictionary $\mathcal{E}$. Each transformation (initial, elementary, or final) of the algorithm *InsertEdge* can be performed in $O(\log n)$ time by means of $O(1)$ link/cut and expand/contract operations on tree $\mathcal{T}^*$ and $O(1)$ updates (insertions or deletions) of $\mathcal{E}$. Hence the time complexity of operation *InsertEdge* is $O((1+T)\log n)$, where T is the number of transformations performed.

THEOREM 6.5. *The amortized time complexity of operation InsertEdge over a sequence of update operations is* $O(\log n)$.

Proof. Let R, S, and P denote the sets of R-, S-, and P-nodes of $\mathcal{T}$, respectively, and let $\deg(\mu)$ denote the number of children of node μ. We define the following *potential function* associated with the data structure:

$$\Phi = |R| + \sum_{\mu \in S \cup P} \deg(\mu).$$

Define the amortized number of transformations performed by *InsertEdge* as $A = T + \Delta\Phi$, where $\Delta\Phi$ is the variation of potential. An RR-transformation decreases by one the number of R-nodes and does not change the degrees of S- and P-nodes. An RP-transformation does not change the number of R-nodes and decreases by one the sum of the degrees of S- and P-nodes. An RP-transformation does not change the number of R-nodes and decreases by one the sum of the degrees of S- and P-nodes. The initial and final transformations in *InsertEdge* change the potential by a constant. Hence, we conclude that $A = O(1)$. Since $|\Phi| = O(n)$, the total time complexity of a sequence of n update operations starting from a graph with $O(1)$ vertices is $O(n \log n)$. □

We conclude the following.

THEOREM 6.6. *There exists a data structure for on-line planarity testing of a biconnected planar graph G whose current number of vertices is n with the following performance: the space requirement is $O(n)$; operations Test and InsertVertex take worst-case time $O(\log n)$, and operation InsertEdge takes amortized time $O(\log n)$.*

Proof. The space and time complexity bounds follow immediately from Lemmas 6.3 and 6.4 and Theorem 6.5. □

7. Tests and updates in general graphs.

In this section, we consider on-line planarity testing for general (nonbiconnected) planar graphs. We first consider connected graphs and then disconnected graphs.

7.1. Tests in connected graphs.

We consider a connected planar graph G with n vertices. We use the data structure of the previous section for each block (biconnected component) of G and represent the relationship between blocks by means of the block-cutvertex tree.

The *block-cutvertex tree* of a connected graph G has a B-node for each block (biconnected component) of G, a C-node for each cutvertex of G, and edges connecting each B-node μ to the C-nodes associated with the cutvertices in the block of μ (see, e.g., [28]). The block-cutvertex tree was previously used in [53, 60] for maintaining biconnected components.

We construct an augmented block-cutvertex tree $\mathcal{B}$ for G as follows (see Fig. 19). We root $\mathcal{B}$ at an arbitrary B-node. Next, we add n new leaf nodes, called V-nodes, to $\mathcal{B}$, each associated with a vertex of G. The parent of the V-node representing vertex v it is the C-node associated with v if v is a cutvertex, and is the B-node associated with the unique block containing v otherwise. The number of nodes of $\mathcal{B}$ is $O(n)$. We store at each B-node μ a secondary structure consisting of the data structure of the previous section for on-line planarity testing in the block B of μ.

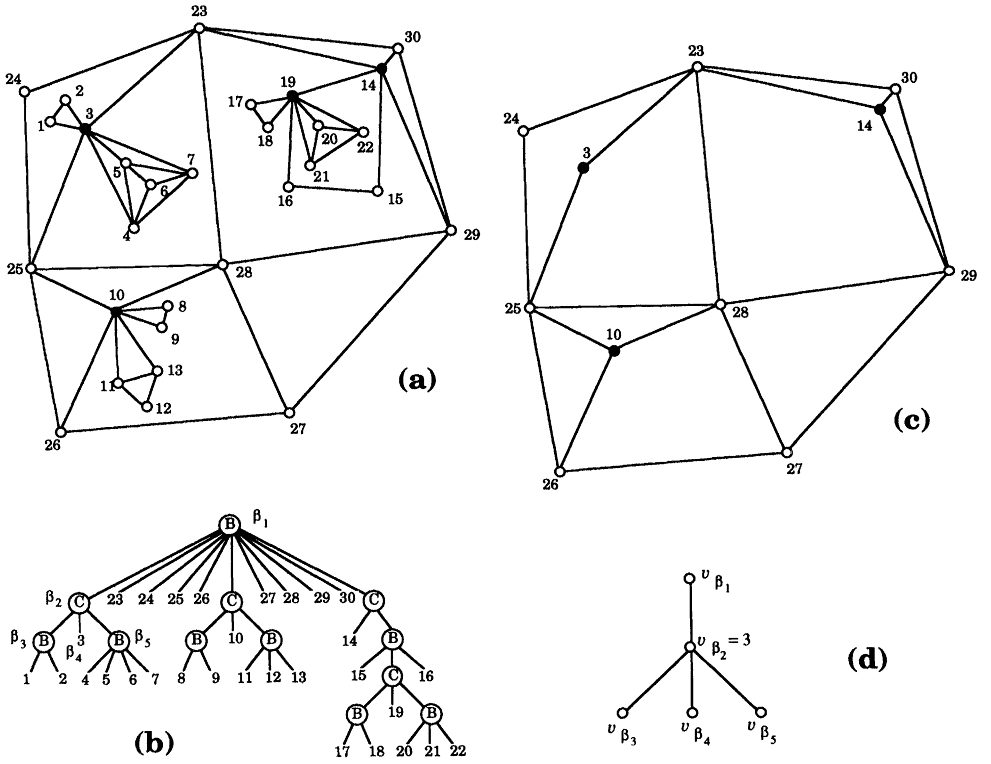


FIG. 19. (a) *A 1-connected graph G.* (b) *The augmented block-cutvertex tree $\mathcal{B}$ of G.* (c) *skeleton*(β_1). (d) *skeleton*(β_2).

The following definitions are analogous to those given for the decomposition tree in §4.

We define graph *skeleton*(μ) for a node μ of $\mathcal{B}$ as follows (see Fig. 19):

- If μ is a V-node representing vertex v, then *skeleton*(μ) consists of a single vertex v.
- If μ is a B-node, then *skeleton*(μ) is the block corresponding to μ. Each child ν of μ is a V- or C-node and hence uniquely associated with a vertex v. The virtual vertex of ν in *skeleton*(μ) is v.
- If μ is a C-node, let c be the cutvertex associated with μ and k be the number of children of μ in $\mathcal{B}$; *skeleton*(μ) is a "star" tree with $k+2$ vertices, where the center vertex is c and the other vertices are the virtual vertices of the

children of μ, plus a vertex representing the parent of μ.

Note that each child of μ is uniquely associated with a vertex of $\mathit{skeleton}(\mu)$. It is easy to see that two vertices are in the same block if and only if the path in $\mathcal{B}$ between their V-nodes has exactly one B-node.

The *pertinent graph* G_μ of a node μ is $\mathit{skeleton}(\mu)$ if μ is a V-node, and it is the union of all the blocks of the B-nodes in the subtree of $\mathcal{B}$ rooted at μ otherwise. The *pivot* of a B-node μ distinct from the root is the cutvertex whose C-node is the parent of μ. The pivot of the C-node of a cutvertex c is c itself. In the example of Fig. 19, vertex 3 is the the pivot of nodes β_2, β_3, β_4, and β_5.

The *expansion graph* of a vertex v of $\mathit{skeleton}(\mu)$ is the pertinent graph of the child of μ with virtual vertex v, or it is v itself if no such child exists. Observe that if v is not a a cutvertex of G, then its expansion graph is v itself.

A vertex v is *pivotal* in the pertinent graph G_μ of a node μ if it appears in the same face of the pivot of μ in some embedding of G_μ. We say that a node ν is *pivotal* if the pivot of ν is pivotal in the pertinent graph of the parent of ν. Note that a child of a C-node is always pivotal. In the example of Fig. 19, the V-node of vertex 5 is pivotal while the V-node of vertex 6 is not pivotal.

Let v be a vertex of the pertinent graph G_μ of a node μ of $\mathcal{B}$. The *representative* of v in the skeleton of μ is the vertex x of $\mathit{skeleton}(\mu)$ defined as follows: if v is in $\mathit{skeleton}(\mu)$, then $x = v$; otherwise, x is the vertex of $\mathit{skeleton}(\mu)$ whose expansion graph contains v. In the example of Fig. 19, the representative of vertex 5 is vertex 3 in $\mathit{skeleton}(\beta_1)$ and is vertex v_{β_5} in $\mathit{skeleton}(\beta_2)$.

We now show how to perform operation $\mathit{Test}(v_1, v_2)$ for vertices in distinct blocks (see Fig. 19).

THEOREM 7.1. *Let v_1 and v_2 be vertices of a connected planar graph G. There exists an embedding Γ of G such that v_1 and v_2 are on the same face of Γ if and only if there exists a node χ of the block-cutvertex tree $\mathcal{B}$ of G such that*

1. *v_1 and v_2 have distinct representatives x_1 and x_2 in χ;*
2. *x_1 and x_2 are on the same face of some embedding of $\mathit{skeleton}(\chi)$; and*
3. *v_1 and v_2 are pivotal vertices of the expansion graphs of x_1 and x_2, respectively.*

Proof. The proof is essentially the same as that of Theorem 4.6, except for the different meaning of the terminology. □

We provide now examples of application of Theorem 7.1 referring to the graph of Fig. 19. Regarding $\mathit{Test}(1, 4)$, we have $v_1 = 1$, $v_2 = 4$, $\chi = \beta_2$, $x_1 = v_{\beta_3}$, and $x_2 = v_{\beta_5}$, so that all the conditions of Theorem 7.1 are verified. Regarding $\mathit{Test}(5, 20)$, we have $v_1 = 5$, $v_2 = 20$, $\chi = \beta_1$, $x_1 = 3$, and $x_2 = 14$, and again all the conditions of the theorem are verified. Indeed, see in Fig. 20 how edge $(5, 20)$ can be inserted. Regarding $\mathit{Test}(6, 20)$, we have $v_1 = 6$, $v_2 = 20$, $\chi = \beta_1$, $x_1 = 3$, and $x_2 = 14$, and condition 3 is not verified since v_1 is not pivotal in the expansion graph of x_1. Finally, regarding $\mathit{Test}(12, 20)$, we have $v_1 = 12$, $v_2 = 20$, $\chi = \beta_1$, $x_1 = 10$, and $x_2 = 14$, and condition 2 is not verified since x_1 and x_2 are not on the same face of some embedding of $\mathit{skeleton}(\chi)$.

The following lemma will be used to efficiently test condition 3.

LEMMA 7.2. *Vertex v is pivotal in G_χ if and only if the first nonpivotal node on the path of $\mathcal{B}$ from the V-node of v to the root is either a child of χ or a node of the path from χ to the root.*

It is interesting to observe the analogy between the concepts of peripheral (defined in §4) and pivotal. The proof of Lemma 7.2 is analogous to that of Lemma 4.5.

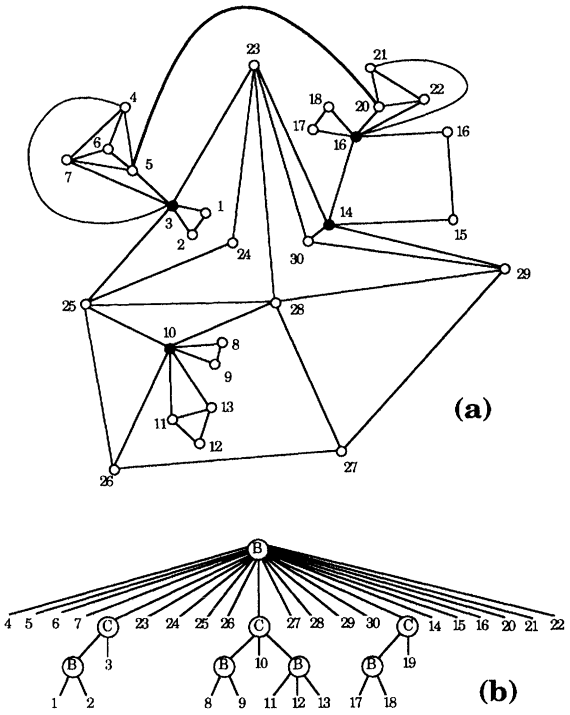


FIG. 20. *Example of operation InsertEdge for vertices in distinct blocks:* (a) *graph of Fig.* 19 *after performing operation InsertEdge*$(e, 5, 20)$; (b) *its augmented block-cutvertex tree.*

THEOREM 7.3. *Let G be a connected planar graph with n vertices. There exists an $O(n)$-space data structure that supports operation Test(u, v) on G in time $O(\log n)$ and can be constructed in $O(n)$ preprocessing time.*

Proof. Condition 1 of Theorem 7.1 is verified for at most two nodes of $\mathcal{B}$: always for the least common ancestor μ of the V-nodes of v_1 and v_2 and possibly for a child of μ. The latter case arises when v_1 is a cutvertex and v_2 is in a block whose B-node is a child of the C-node of v_1. Condition 2 is equivalent to performing operation *Test*(x_1, x_2) in *skeleton*(χ), which is either a biconnected graph or consists of a single vertex.

By Lemma 7.2, we can test condition 3 of Theorem 7.1 using for each vertex v a pointer to the first node η in the path from the V-node of v to the root of $\mathcal{B}$ such that v is not pivotal in G_η.

The time and space complexity bounds follow from Theorem 4.7 and from the fact that tree $\mathcal{B}$ has $O(n)$ nodes. □

COROLLARY 7.4. *Let G be a connected planar graph with n vertices. There exists an $O(n)$-space data structure that supports operation Test(u, v) on G in time $O(1)$ and can be constructed in $O(n^2)$ preprocessing time.*

7.2. Updates in connected graphs. We consider a connected planar graph G with n vertices. It is easy to see that operations *InsertEdge* and *AttachVertex* form a complete repertoire of operations for connected planar graphs.

LEMMA 7.5. *A connected planar graph G with n vertices and m edges can be assembled starting from a single vertex by means of m InsertEdge and AttachVertex* *operations, such that each intermediate graph is planar and connected. Also, such a sequence of operations can be determined in $O(n)$ time.*

Proof. First, construct a spanning tree of G by means of $n - 1$ *AttachVertex* operations, and then add the remaining edges with $m - n + 1$ *InsertEdge* operations. □

Operation $AttachVertex(e, v, u)$ is performed as follows. First, if u is a cutvertex, let β' be its C-node; otherwise, replace the V-node of u with a new C-node β' and a child V-node. Second, create a new B-node β'' (associated with the block consisting of the newly added edge e) with a child V-node (associated with the newly added vertex v) and make β'' a child of β'.

We now examine the structural changes of the block-cutvertex tree when operation $InsertEdge(e, v_1, v_2)$ is performed on G. If v_1 and v_2 are in the same block B of G, then the primary structure of the block-cutvertex tree stays unchanged, and we process the insertion in the secondary structure (decomposition tree) of the B-node of B. Otherwise (see Fig. 20), let μ_1 and μ_2 be the V-nodes of v_1 and v_2, respectively, and let χ the least common ancestor of μ_1 and μ_2. The effect of *InsertEdge* is to merge the "old" blocks corresponding to the B-nodes of $\mathcal{B}$ on the paths of $\mathcal{B}$ from μ_1 to χ and from μ_2 to χ (inclusive) into a "new" block B'.

The primary structure of $\mathcal{B}$ is updated by means of a sequence of primitive tree operations (see §6.3). To update the secondary structure, we need to efficiently merge the decomposition trees of the old blocks into the decomposition tree of the new block B'. We reorient all the old blocks, except the largest one, denoted B^*. Avoiding the reorientation of B^* is the key to the efficient amortized behavior of the *InsertEdge* algorithm. Each old block distinct from B^* is reoriented into a planar *st*-graph with poles given by consecutive cutvertices in the chain from μ_1 to μ_2. By adding these orientations to the orientation of B^*, we obtain a planar *st*-orientation of the B' whose poles are the same as those of B^*. See a schematic example in Fig. 21.

The decomposition tree of the new block B' is obtained as follows. Let β_1 and β_2 be the nodes of $\mathcal{B}$ adjacent to the B-node of B^* in the path of $\mathcal{B}$ between the V-nodes of v_1 and v_2, with β_i on the side of v_i, $i = 1, 2$. Let u_i be the vertex associated with node β_i, $i = 1, 2$. (Note that β_i is either a C-node or a V-node.) We perform $InsertEdge(e, u_1, u_2)$ in B^* and then replace the Q-node of e in the decomposition tree of B^* with an S-node whose subtrees are the newly built decomposition trees of the other old blocks. Note that one of u_1 and u_2 is the former pivot c^* of B^*.

In our dynamic environment, we maintain the forest $\mathcal{P}$, called *pivotal forest*, obtained from $\mathcal{B}$ by removing all the edges from nodes that are not pivotal to their parents. Hence the first nonpivotal node on the path of $\mathcal{B}$ from the V-node of v to the root (see Lemma 7.2) is the root of the tree of $\mathcal{P}$ containing the V-node of v.

To update the pivotal forest, we observe that the pivot c of the new block B' is the cutvertex parent of χ and is in general (when B^* is not the block of χ) different from the former pivot c^* of B^* (see Fig. 21). Let μ be the proper allocation node of c^* in the decomposition tree of B'. The new pivot c is in the pertinent graph of an edge of $skeleton(\mu)$ incident upon c^*. Such a pertinent graph is an orientation with poles u_1 and u_2 of the union of edge e and the old blocks except B^*. Hence we have that nodes of $\mathcal{B}$ can go from pivotal to nonpivotal but not vice versa.

To efficiently maintain the pivotal forest $\mathcal{P}$, we use the following auxiliary data structure. Consider a B-node of $\mathcal{P}$ associated to a block B, and let c be its pivot. Let $\mathcal{T}(B)$ be a new copy of the decomposition tree of B associated with a planar *st*-orientation of B with $t = c$. We modify $\mathcal{T}$ into a forest $\mathcal{P}^*(B)$ as follows:

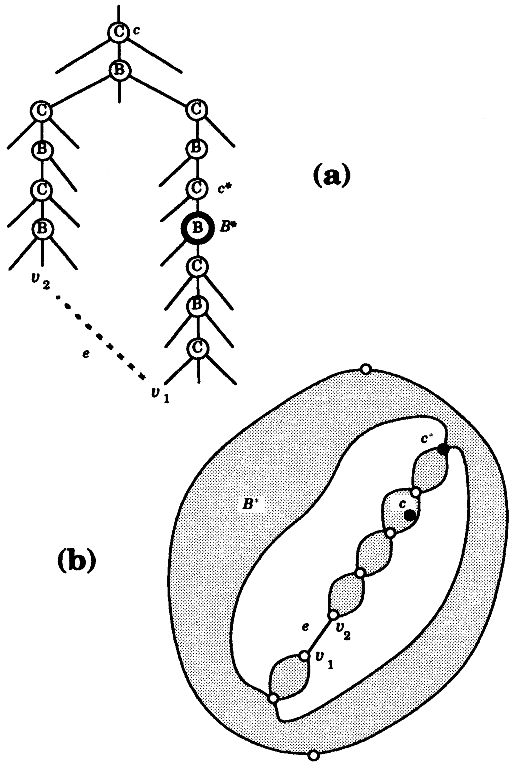


FIG. 21. *Schematic example of operation InsertEdge for vertices in distinct blocks:* (a) *augmented block-cutvertex tree*; (b) *reorientation of the blocks, except the largest one, denoted* B^*.

1. Let $\mathcal{P}^*(B) = \mathcal{T}(B)$.
2. Remove from $\mathcal{P}^*(B)$ the Q-nodes.
3. For each (remaining) node μ of $\mathcal{P}^*(B)$, if μ is nonperipheral and the virtual edge of μ is not in the same face as c in *skeleton*(ν), we remove the edges from μ to its parent ν.
4. For each vertex $u \neq c$, create a U-node μ. Let ν be the proper allocation node of u and (s_ν, t_ν) be the edge between the poles of ν. Make μ a child of ν if one of the following cases applies: (i) c is in *skeleton*(ν), and c and u are on a common face of *skeleton*(ν); or (ii) c is not in *skeleton*(ν), and (s_ν, t_ν) and u are on a common face of *skeleton*(ν).
5. For each allocation node ν of c, let $e_0 = (c, u_0), \ldots, e_{k-1} = (c, u_{k-1})$ be the edges incident on c in *skeleton*(ν) in clockwise order around c; μ_i be the child of ν whose virtual edge in *skeleton*(ν) is e_i; f_i $(i = 0, \ldots, k-1)$ be the face of *skeleton*(ν) containing edges e_i and $e_{(i+1) \bmod k}$; and E_i be the set of edges of f_i excluding e_i and $e_{(i+1) \bmod k}$.
   (a) Expand node ν into a node ν' with k new children $\nu_0, \ldots, \nu_{k-1}$.
   (b) For $i = 0, \ldots, k-1$, make μ_i and the U-node κ_i of u_i be children of ν'.
   (c) Let the order of the children of ν' from left to right be $\mu_0, \kappa_0, \nu_0, \ldots, \mu_{k-1}, \kappa_{k-1}, \nu_{k-1}$.
   (d) For $i = 0, \ldots, k-1$, make each former child of ν whose virtual edge is in E_i be a child of ν_i

   (e) For $i = 0, \ldots, k-1$, make each U-node that is a former child of ν whose associated vertex u is in face f_i and is not adjacent to c a child of ν_i.
   (f) For $i = 0, \ldots, k-1$, order the children of ν_i from left to right according to the clockwise sequence of vertices and edges in face f_i of *skeleton*(ν).
6. For each node ν that is not an allocation node of c, let f_0 and f_1 be the faces on the two sides of the edge between the poles of ν (s_ν, t_ν) in *skeleton*(ν), and let E_i $(i = 0, 1)$ be the set of edges of f_i excluding (s_ν, t_ν).
   (a) Expand node ν into a node ν' with children ν_0 and ν_1.
   (b) For $i = 0, 1$, make each former child of ν whose virtual edge is in E_i be a child of ν_i
   (c) For $i = 0, 1$, make each U-node that is a former child of ν whose associated vertex u is in face f_i a child of ν_i.
   (d) For $i = 0, 1$, order the children of ν_i from left to right according to the clockwise sequence of vertices and edges in face f_i of *skeleton*(ν).

We replace the B-node of block B in $\mathcal{B}$ with forest $\mathcal{P}^*(B)$ and identify each V- and C-node that is a former child of the B-node with the U-node of $\mathcal{P}^*(B)$ associated with the same vertex. We denote by $\mathcal{P}^*$ the resulting forest. The correspondence between $\mathcal{P}$ and $\mathcal{P}^*$ is given by the following lemma.

LEMMA 7.6. *Let ν be the B-node of $\mathcal{P}$ associated with a block B and μ be the child of ν associated with a vertex u of B. There is an edge in $\mathcal{P}$ between μ and ν (i.e., u is pivotal in B) if and only if there is a path in $\mathcal{P}^*(B)$ between the U-node of u and the root of $\mathcal{P}^*(B)$.*

Proof. The proof follows from Lemma 7.2 and the above construction. □

By Theorem 7.1 and Lemmas 7.2 and 7.6, forest $\mathcal{P}^*$ is equivalent to $\mathcal{P}$ regarding *Test* operations. We represent forest $\mathcal{P}^*$ with an edge-ordered dynamic tree [15]. Hence operation *Test* can be performed in $O(\log n)$ time.

Now we examine how to modify forest $\mathcal{P}^*$ in consequence of update operations on G. We consider operation *InsertEdge* first for vertices in the same block and then for vertices in distinct blocks. We omit the discussion of operations *InsertVertex* and *AttachVertex*.

When operation *InsertEdge* joins vertices in the same block B, the modifications of $\mathcal{P}^*$ are in exact correspondence with the transformations performed on the decomposition tree of B and take additional $O(\log n)$ time (amortized).

When operation *InsertEdge* joins vertices in distinct blocks, referring to the terminology developed earlier in this section, we construct $\mathcal{P}^*(B')$ by rebuilding $\mathcal{P}^*(B)$ for each old block B except the largest block B^* and by restructuring $\mathcal{P}^*(B^*)$ as follows:

1. Restructure $\mathcal{P}^*(B^*)$ in consequence of *InsertEdge*(e, u_1, u_2). Without loss of generality, assume that $c^* = u_1$.
2. Let T be the tree of $\mathcal{P}^*(B^*)$ containing the U-node κ_2 of u_2, and let ρ be the root of T.
3. Reroot T at node κ_2.
4. Perform local updates along the path of T from ρ to κ_2.
5. Link T to the the rest of the newly reconstructed $\mathcal{P}^*(B')$.

Let ℓ be the length of the path between ρ and κ_2, and let $\Delta_{\mathcal{P}}$ be the variation of the number of edges of $\mathcal{P}$ in consequence of *InsertEdge*. We have the following result.

LEMMA 7.7. *The restructuring of $\mathcal{P}^*$ takes time $O(\log n - \Delta_{\mathcal{P}} + \ell)$.*

Proof. The rerooting ot T and local updates are performed using a variation of operation *evert* of edge-ordered dynamic trees. This takes $O(\log n)$ time plus $O(-\Delta_{\mathcal{P}} + \ell)$

time to perform the local updates. □

We conclude this section with the amortized analysis of the time complexity of operation *InsertEdge* for vertices in distinct blocks. Let $\mathcal{O}$ be the set of old blocks. Denote by n_B the number of vertices of block B and by $\Delta_{\mathcal{B}}$ the variation of the number of nodes of $\mathcal{B}$ in consequence of *InsertEdge*. (Note that $\Delta_{\mathcal{B}} \geq -2 \cdot |\mathcal{O}| + 1$.) Let t^* be the time to perform *InsertEdge*(e, u_1, u_2) in B^*.

From the above discussion, we have that, with an appropriate choice of the time unit, the total time t for operation *InsertEdge* is

$$t = t^* + t_{\mathcal{B}} + t_{\mathcal{O}} + t_{\mathcal{P}} + \log n,$$

where

$$\begin{aligned} t_{\mathcal{O}} &= \sum_{B \in \mathcal{O} - \{B^*\}} n_B, \\ t_{\mathcal{B}} &= -\Delta_{\mathcal{B}} \cdot \log n, \\ t_{\mathcal{P}} &= -\Delta_{\mathcal{P}} + \ell. \end{aligned}$$

Namely, $t_{\mathcal{O}}$ is the time to reorient the old blocks and rebuild their secondary structures; $t_{\mathcal{B}}$ is the time to update the primary structure of $\mathcal{B}$ by means of primitive tree operations; and $t_{\mathcal{P}}$ is the time to update forest $\mathcal{P}^*$.

Let $\mathcal{A} = \bigcup_B \mathcal{A}_B$, where $\mathcal{A}_B$ is the set of allocation nodes of the cutvertex-c parent of B in $\mathcal{B}$, in the decomposition tree of B. Let $|\mathcal{P}|$ denote the total number of edges of $\mathcal{P}$. Let Φ_B be the potential of the data structure for block B, as given in the proof of Theorem 6.5. We define the potential Φ of the data structure as

$$\Phi = \sum_B \left[n_B \cdot \log \frac{1}{n_B} + (\Phi_B + a + 2) \cdot \log n \right] + |\mathcal{P}| + |\mathcal{A}|,$$

where the constant $a > 0$ denotes the maximum variation of potential of a block in consequence of an *InsertEdge* operation. (Recall that the proof of Theorem 6.5 shows that such variation is bounded by a positive constant.) Since $|\mathcal{P}| + |\mathcal{A}| = O(n)$, we have that $|\Phi| = O(n \log n)$.

Denoting by $\Delta_{\mathcal{A}}$ the variation of cardinality of $\mathcal{A}$, the variation $\Delta\Phi$ of potential in consequence of *InsertEdge* is given by

$$\Delta\Phi = \Delta\Phi_{\mathcal{O}} + \Delta\Phi_{\mathcal{B}} + \Delta\Phi_{\mathcal{P}},$$

where

$$\begin{aligned} \Delta\Phi_{\mathcal{O}} &= n_{B'} \cdot \log \frac{1}{n_{B'}} - \sum_{B \in \mathcal{O}} \left(n_B \cdot \log \frac{1}{n_B} \right), \\ \Delta\Phi_{\mathcal{B}} &= (\Phi_{B'} + a + 2) \cdot \log n - \sum_{B \in \mathcal{O}} (\Phi_B + a + 2) \cdot \log n, \\ \Delta\Phi_{\mathcal{P}} &= \Delta_{\mathcal{P}} + \Delta_{\mathcal{A}}. \end{aligned}$$

The following lemma is proved in [12]. Its proof is sketched here for the reader's convenience.

LEMMA 7.8 (see [12]). *Consider the function $f(x) = x \log \frac{1}{x}$, and let $1 \leq x_1 \leq \cdots \leq x_k$. We have*

$$f(x_1 + \cdots + x_k) - (f(x_1) + \cdots + f(x_k)) \leq -2(x_1 + \cdots + x_{k-1}).$$

Proof. This is a proof by induction on k. The base case ($k = 2$) is easy to prove by a simple analysis of the binary entropy function $h(x) = f(x) + f(1-x)$ for $0 < x < 1$. □

By Lemma 7.8, $t_{\mathcal{O}} + \Delta\Phi_{\mathcal{O}} \leq 0$. By Theorem 6.5, we have

$$t^* = -\Delta\Phi_{B^*} \cdot \log n + \log n$$

and

$$\Phi_{B'} \leq \sum_{B \in \mathcal{O}} (\Phi_B + a) + \Delta\Phi_{B^*}.$$

Thus

$$t^* \leq -\Phi_{B'} \cdot \log n + \sum_{B \in \mathcal{O}} (\Phi_B + a) \cdot \log n.$$

This implies that $t^* + t_{\mathcal{B}} + \Delta\Phi_{\mathcal{B}} = O(\log n)$. Finally, we have that $\ell \leq -\Delta_{\mathcal{A}}$, and therefore $t_{\mathcal{P}} + \Delta\Phi_{\mathcal{P}} \leq 0$. We conclude that $t + \Delta\Phi = O(\log n)$, so that the amortized time complexity of *InsertEdge* is $O(\log n)$.

7.3. Disconnected graphs. For graphs that are not connected, we complete our repertoire with operation *MakeVertex*, which can clearly be performed in $O(1)$ time. Let G be a general planar graph (possibly disconnected). We consider the *block-cutvertex forest* of G, which is the forest of the block-cutvertex trees of the connected components of G. When an *InsertEdge* operation joins two old connected components C' and C'' into a new connected component C, we rebuild the block-cutvertex tree of the smaller old component, so that it can be linked as a subtree of the block-cutvertex tree of the larger component.

Again, we perform an amortized analysis. Let n_C denote the size of the connected component C. The time for *InsertEdge* is $t = \log n + \min(n_{C'}, n_{C''})$. The potential of the data structure is defined as

$$\Phi = \sum_{\text{all components } C} \left(b \cdot n_C \cdot \log \frac{1}{n_C} + \Phi_C \right),$$

where $b > 1$ is a constant such that $|\mathcal{P}| + |\mathcal{A}| \leq b \cdot n$, and Φ_C is the potential of connected component C as defined in §7.2. We can immediately verify that joining C' and C'' affects the part of the potential associated with the size of $\mathcal{P}$ and $\mathcal{A}$ so that it increases by at most $b \cdot \min(n_{C'}, n_{C''})$. By Lemma 7.8, we conclude that $t + \Delta\Phi = O(\log n)$.

The above analysis concludes the proof of Theorem 1.1 stated in §1.

8. Some applications.

8.1. Graph planarization. Let G be a graph with n vertices and m edges. Given a set of weights on the edges of G, a *maximum-weight planar subgraph* of G is a planar subgraph of G with the maximum total edge weight. Finding a maximum-weight planar subgraph is NP-hard even if all the edges have unit weights [26]. Given an ordering $e_0, \ldots, e_{m-1}$ of the edges of G, each subgraph S of G is identified by an integer $k(S) = (b_{m-1} \cdots b_0)_2$ such that bit $b_i = 1$ if and only if edge e_i is in S. The *lexicographically maximum planar subgraph* of G with respect to the given edge ordering is the planar subgraph S of G with maximum $k(S)$. Several heuristics have been developed for computing maximum-weight planar subgraphs; see, e.g., [14, 41]. There is experimental evidence that if the edges are sorted by decreasing weight, a lexicographically maximum planar subgraph provides a good approximation of a maximum-weight planar subgraph [14]. As a corollary of Theorem 1.1, we can construct a lexicographically maximum planar subgraph in $O(m \log n)$ time. The same result has been obtained in [7] with a different technique. A maximal planar subgraph of G can instead be constructed in $O(n+m)$ time [49] using the technique of [20].

8.2. Transitive closure. In this section, we study the maintenance of reachability information in planar st-graphs, an important class of planar digraphs that find several applications in computational geometry [27, 44, 45] and graph layout [10, 13]. Queries are of the following type: "Is there a directed path from v_1 to v_2?" This problem was previously best solved using a data structure for general digraphs with $O(n^2)$ space, $O(1)$ query time, and $O(n)$ amortized update time [32, 43].

However, this problem admits an efficient algorithm if we assume that the digraph is embedded and restrict the *InsertEdge* operation to join vertices on the same face of the embedding. That is, in the fixed-embedding problem, an edge that preserves planarity cannot be added if this requires a change of the embedding. A data structure for the fixed-embedding version of reachability in planar st-graphs is presented in [54]. The space requirement is $O(n)$, and the query/update time is $O(\log n)$ (worst-case). We show that the fixed-embedding restriction of the above technique can be removed by using the decomposition tree (see §4) to maintain a hierarchical representation of the embedding.

More formally, the *on-line reachability problem* for a planar st-graph G consists of performing on G a sequence of update operations $\mathit{InsertEdge}(e, v_1, v_2)$ and $\mathit{InsertVertex}(v_1, v_2)$, intermixed with queries of the following type:

$\mathit{Reachable}(v_1, v_2)$: Determine whether there exists a directed path from v_1 to v_2.

It is shown in [54] that an embedded planar st-graph admits two total orderings of its vertices, edges, and faces, called *left-sequence* and *right-sequence*, such that there exists a path from v_1 to v_2 if and only if v_1 precedes v_2 in both the left- and right-sequences. The update of the left- and right sequences after an *InsertEdge* operation that preserves the embedding consists of a simple exchange of subsequences, so that it can be efficiently supported by means of concatenable queues. See [54] for details.

The extension of the technique of [54] to *InsertEdge* operations that arbitrarily modify the embedding is based on the following properties, whose proof is left to the reader.

LEMMA 8.1. *Let μ be a node of the decomposition tree of a planar st-graph G. The left-sequence (right-sequence) of the pertinent graph of μ can be obtained from the left-sequence (right-sequence) of skeleton(μ) by replacing each edge e with the left-sequence (right-sequence) of the expansion graph of e minus its poles.*

LEMMA 8.2. *Flipping the embedding of a planar st-graph around the poles exchanges the left-sequence with the right-sequence.*

We consider an (arbitrary) embedding of graph G, and we maintain two copies of the decomposition tree, denoted $\mathcal{T}_L$ and $\mathcal{T}_R$, which differ only in the order of the children at each node. In tree $\mathcal{T}_L$, the children of each node are ordered according to the left-sequence of the corresponding virtual edges. Tree $\mathcal{T}_R$ is similarly defined with respect to the right-sequence. There is a one-to-one correspondence between the nodes of $\mathcal{T}_L$ and $\mathcal{T}_R$. By Lemma 8.1, the left-to-right order of the Q-nodes of $\mathcal{T}_L$ ($\mathcal{T}_R$)

yields the subsequence of edges of the left-sequence (right-sequence). Since vertices are not explicitly described in our representation of the left- and right-sequences, we perform the query *Reachable*(v_1, v_2) by considering any edge e_1 incoming to v_1 and any edge e_2 outgoing from v_2, and we test whether e_1 precedes e_2 in both the left- and right-sequence. If v_1 has no incoming edges, then it must be the source of G, and hence it reaches v_2. A similar argument applies if v_2 has no outgoing edges. If *Reachable*$(v_1, v_2) =$ *true*, a path from v_1 to v_2 can be reported in time $O(k)$ with a visit of the decomposition tree, where k is the path length.

When adding an edge to G, we may have to modify the embedding. This is done by means of the primitive topological operations reverse and swap; see Lemma 4.2. A reverse operation flips the embedding of the pertinent graph of a node around its poles. A swap operation restructures the embedding of the pertinent graph of a P-node μ by embedding the pertinent graph of a child of μ in a different position. By Lemma 8.2 the reverse operation corresponds to exchanging the subtrees of $\mathcal{T}_L$ and $\mathcal{T}_R$ rooted at the corresponding nodes. It is performed by two cuts followed by two links. The swap operation is performed by means of two link and cut operations at two corresponding P-nodes of $\mathcal{T}_L$ and $\mathcal{T}_R$.

After the embedding has been modified so that v_1 and v_2 are on the same face, the exchange of subsequences in the left-sequence and right-sequence caused by *InsertEdge*(e, v_1, v_2) is performed only at node χ associated with v_1 and v_2 (see *InsertEdge* in Algorithm 2), and can be done with $O(1)$ primitive tree operations. We represent $\mathcal{T}_L$ and $\mathcal{T}_R$ as ordered dynamic trees [15].

THEOREM 8.3. *Let G be a planar st-graph with n vertices. There exists an $O(n)$-space data structure for the on-line reachability problem in G that supports operations Reachable, InsertEdge, and InsertVertex in $O(\log n)$ time, where the bound is amortized for InsertEdge and worst-case for the other operations. Also, a directed path between two vertices can be reported in time $O(\log n + k)$, where k is the path length.*

8.3. Minimum spanning tree. In this section, we investigate the maintenance of a minimum spanning tree of a planar graph under weight changes and insertions of vertices and edges. Queries are of the following type: "Is edge e in the current minimum spanning tree?" The previous best result for this problem is an $O(m)$-space data structure for general graphs supporting queries in $O(1)$ time and updates in $O(\sqrt{m})$ time [21]. Also, this problem admits a more efficient algorithm if we assume that the graph is embedded and restrict the *InsertEdge* operation to join vertices on the same face of the embedding. Namely, $O(n)$ space and $O(\log n)$ query/update time (worst-case) can be achieved [15].

The *on-line minimum-spanning-tree problem* consists of maintaining the minimum spanning tree of a graph. First, we consider the case of a biconnected planar graph that is subject to a sequence of updates *InsertVertex* and *InsertEdge*, intermixed with the following operations:

InMst(e): Determine whether edge e belongs to the current minimum spanning tree.

Reweight(e, w): Set the weight of edge e equal to w.

The fixed-embedding technique of [15] is based on the fact that the edges of G not in the minimum spanning tree T dualize to a maximum spanning tree T^* of the dual graph G^*. The cocycle (partition of the vertices) induced by the deletion of edge e from T corresponds to the cycle induced by the insertion of e^* into T^*, and vice versa. Hence the dynamization of the minimum spanning tree can be done by representing both T and T^* by ordered dynamic trees that support the usual tree operations plus queries on the minimum-/maximum-weight edge on the tree path between two vertices.

As previously described, the decomposition tree can be supplemented with embedding information so that the modifications of the embedding required by an *InsertEdge* operation are carried out in amortized time $O(\log n)$. Regarding the update of the dual tree T^*, we observe that the faces to the left and right of a pertinent graph are a separation pair of G^*. Hence T^* is updated in a reverse or swap operation by means of a sequence of $O(1)$ expand, cut, link, and contract operations performed at the nodes of T^* representing the faces to the left and right of the pertinent graph being flipped or moved. Figure 22 shows a schematic example of the update of T^* in a reverse operation.

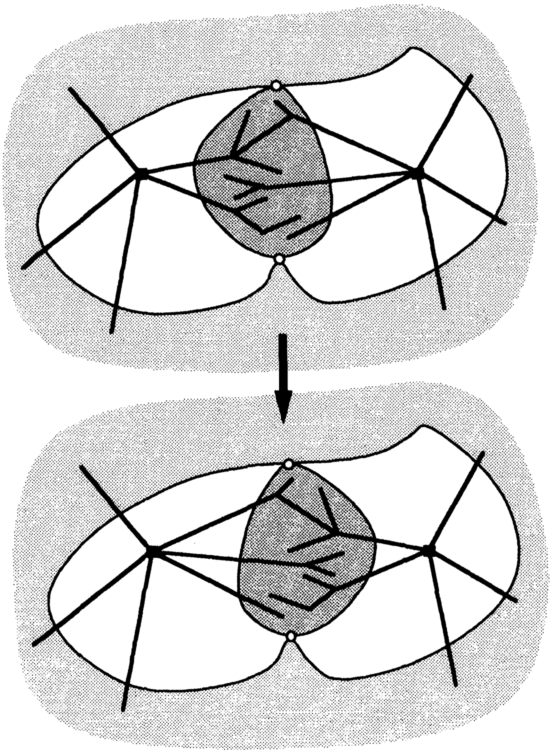


FIG. 22. *Schematic example of the update of the dual tree T^* in a reverse operation. The pertinent graph being reversed and the portion of T^* in it are shown. Dual nodes are represented by filled squares, and dual edges are represented by thick lines.*

THEOREM 8.4. *Let G be a planar biconnected graph with n vertices. There exists a data structure for the on-line minimum-spanning-tree problem that supports operation InMst in $O(1)$ time and operations InsertVertex, InsertEdge, and Reweight in $O(\log n)$ time. The time bound is amortized for InsertEdge and worst-case for the other operations.*

The minimum spanning tree of a nonbiconnected graph is the union of the minimum spanning trees of its blocks. Hence we have the following result.

THEOREM 8.5. *Let G be a planar graph with n vertices. There exists a data structure for maintaining on-line a minimum spanning forest of G that supports operation InMst in $O(1)$ time and operations InsertVertex, MakeVertex, InsertEdge, and*

Reweight in $O(\log n)$ time. The time bound is amortized for InsertEdge and worst-case for the other operations.

REFERENCES

[1] B. Alpern, R. Hoover, B. Rosen, P. Sweeney, and F. K. Zadeck, *Incremental evaluation of computational circuits*, in Proc. ACM–SIAM Symposium on Discrete Algorithms, Society for Industrial and Applied Mathematics, Philadelphia, 1990, pp. 32–42.

[2] G. Ausiello, G. F. Italiano, A. Marchetti-Spaccamela, and U. Nanni, *Incremental algorithms for minimal length paths*, in Proc. ACM–SIAM Symposium on Discrete Algorithms, Society for Industrial and Applied Mathematics, Philadelphia, 1990, pp. 12–21.

[3] A. M. Berman, M. C. Paull, and B. G. Ryder, *Proving relative lower bounds for incremental algorithms*, Acta Inform., 27 (1990), pp. 665–683.

[4] D. Bienstock and C. L. Monma, *On the complexity of of covering vertices by faces in a planar graph*, SIAM J. Comput., 17 (1988), pp. 53–76.

[5] K. Booth and G. Lueker, *Testing for the consecutive ones property property, interval graphs, and graph planarity using PQ-tree algorithms*, J. Comput. System Sci., 13 (1976), pp. 335–379.

[6] A. L. Buchsbaum, P. C. Kanellakis, and J. S. Vitter, *A data structure for arc insertion and regular path finding*, in Proc. ACM–SIAM Symposium on Discrete Algorithms, Society for Industrial and Applied Mathematics, Philadelphia, 1990, pp. 22–31.

[7] J. Cai, X. Han, and R. E. Tarjan, *An $O(m \log n)$-time algorithm for the maximal subgraph problem*, SIAM J. Comput., 22 (1993), pp 1142–1162.

[8] 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.

[9] R. F. Cohen and R. Tamassia, *Dynamic expression trees and their applications*, in Proc. ACM–SIAM Symposium on Discrete Algorithms, Society for Industrial and Applied Mathematics, Philadelphia, 1991, pp. 52–61.

[10] G. Di Battista and R. Tamassia, *Algoritms for plane representations of acyclic digraphs*, Theoret. Comput. Sci., 61 (1988), pp. 175–198.

[11] ———, *On-line graphs algorithms with SPQR-trees*, in Automata, Languages, and Programming (Proc. 17th International Colloquium on Automata, Languages, and Programming), Lecture Notes in Comput. Sci. 442, Springer-Verlag, Berlin, New York, Heidelberg, 1990, pp. 598–611.

[12] ———, *On-line maintenance of triconnected components with SPQR-trees*, Algorithmica, 15 (1996), pp. 302–318.

[13] G. Di Battista, R. Tamassia, and I. G. Tollis, *Area requirement and symmetry display in drawing graphs*, in Proc. ACM Symposium on on Computational Geometry, Association for Computing Machinery, New York, 1989, pp. 51–60.

[14] P. Eades, L. Foulds, and J. Giffin, *An efficient heuristic for identifying a maximal weight planar subgraph*, in Combinatorial Mathematics IX, Lecture Notes in Math. 952, Springer-Verlag, Berlin, New York, Heidelberg, 1990, pp. 239–251.

[15] D. Eppstein, G. F. Italiano, R. Tamassia, R. E. Tarjan, J. Westbrook, and M. Yung, *Maintenance of a minimum spanning forest in a dynamic plane graph*, J. Algorithms, 13 (1992), pp. 33–54.

[16] S. Even and H. Gazit, *Updating distances in dynamic graphs*, Methods Oper. Res., 49 (1985), pp. 371–387.

[17] S. Even and Y. Shiloach, *An on-line edge deletion problem*, J. Assoc. Comput. Mach., 28 (1981), pp. 1–4.

[18] S. Even and R. E. Tarjan, *Computing an st-numbering*, Theoret. Comput. Sci., 2 (1976), pp. 339–344.

[19] H. de Fraysseix, J. Pach, and R. Pollack, *Small sets supporting Fary embeddings of planar graphs*, in Proc. 20th ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1989, pp. 426–433.

[20] H. de Fraysseix and P. Rosenstiehl, *A depth-first-search characterization of planarity*, Ann. Discrete Math., 13 (1982), pp. 75–80.

[21] G. N. Frederickson, *Data structures for on-line updating of minimum spanning trees with applications*, SIAM J. Comput., 14 (1985), pp. 781–798.

[22] ———, *Fast algoritms for shortest paths in planar graphs, with applications*, SIAM J. Comput., 16 (1987), pp. 1004–1022.

[23] ———, *Ambivalent data structures for dynamic 2-edge-connectivity and k smallest spanning*

trees, in Proc. 32nd IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1991, pp 632–641.

[24] Z. Galil and G. F. Italiano, *Fully dynamic algorithms for edge-connectivity problems*, in Proc. 23rd ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1991, pp. 317–327.

[25] ———, *Maintaining biconnected components of dynamic planar graphs*, in Automata, Languages, and Programming (Proc. 18th International Colloquium on Automata, Languages, and Programming), Lecture Notes in Comput. Sci. 510, Springer-Verlag, Berlin, New York, Heidelberg, 1991, pp. 339–350.

[26] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman, San Francisco, 1979.

[27] M. T. Goodrich and R. Tamassia, *Dynamic trees and dynamic point location*, in Proc. 23rd ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1991, pp. 523–533.

[28] F. Harary, *Graph Theory*, Addison–Wesley, Reading, MA, 1969.

[29] D. Harel and R. E. Tarjan, *Fast algorithms for finding nearest common ancestors*, SIAM J. Comput., 13 (1984), pp. 338-355.

[30] J. Hopcroft and R. E. Tarjan, *Dividing a graph into triconnected components*, SIAM J. Comput., 2 (1973), pp. 135–158.

[31] ———, *Efficient planarity testing*, J. Assoc. Comput. Mach., 21 (1974), pp. 549–568.

[32] G. F. Italiano, *Amortized efficiency of a path retrieval data structure*, Theoret. Comput. Sci., 48 (1986), pp. 273–281.

[33] ———, *Finding paths and deleting edges in directed acyclic graphs*, Inform. Process. Lett., 28 (1988), pp. 5–11.

[34] G. F. Italiano, A. Marchetti-Spaccamela, and U. Nanni, *Dynamic data structures for series-parallel graphs*, in Algorithms and Data Structures (Proc. 1989 WADS), Lecture Notes in Comput. Sci. 382, Springer-Verlag, Berlin, New York, Heidelberg, 1989, pp. 352–372.

[35] A. Kanevsky, R. Tamassia, J. Chen, and G. Di Battista, *On-line maintenance of the four-connected components of a graph*, in Proc. 32nd IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1991, pp. 793–801.

[36] M.-Y. Kao and P. N. Klein, *Towards overcoming the transitive-closure bottleneck: Efficient parallel algorithms for planar digraphs*, in Proc. 22nd ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1990, pp. 181–192.

[37] P. N. Klein and J. H. Reif, *An efficient parallel algorithm for planarity*, J. Comput. System Sci., 37 (1988), pp. 190–246.

[38] A. Lempel, S. Even, and I. Cederbaum, *An algorithm for planarity testing of graphs*, in Theory of Graphs, International Symposium, Gordon and Breach, New York, 1967, pp. 215–232.

[39] C. C. Lin and R. C. Chang, *On the dynamic shortest path problem*, in Proc. International Workshop on Discrete Algorithms and Complexity, 1989, pp. 203–212.

[40] T. Nishizeki and N. Chiba, *Planar Graphs: Theory and Algorithms*, Ann. Discrete Math. 32, North–Holland, Amsterdam, 1988.

[41] T. Ozawa and H. Takahashi, *A graph-planarization algorithm and its applications to random graphs*, in Graph Theory and Algorithms, Lecture Notes in Comput. Sci. 108, Springer-Verlag, Berlin, New York, Heidelberg, 1981, pp. 95–107.

[42] J. A. La Poutré, *Dynamic graph algorithms and data structures*, Ph.D. thesis, Department of Computer Science, University of Utrecht, Utrecht, The Netherlands, 1991.

[43] J. A. La Poutré and J. van Leeuwen, *Maintenance of transitive closures and transitive reductions of graphs*, in Graph-Theoretic Concepts in Computer Science (Proc. 1987 WG), Lecture Notes in Comput. Sci. 314, Springer-Verlag, Berlin, New York, Heidelberg, 1988, pp. 106–120.

[44] F. P. Preparata and R. Tamassia, *Fully dynamic point location in a monotone subdivision*, SIAM J. Comput., 18 (1989), pp. 811–830.

[45] ———, *Efficient point location in a convex spatial cell complex*, SIAM J. Comput., 21 (1992), pp. 267–280.

[46] V. Ramachandran and J. H. Reif, *An optimal parallel algorithm for graph planarity*, in Proc. 30th IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1989, pp. 282–293.

[47] J. H. Reif, *A topological approach to dynamic graph connectivity*, Inform. Process. Lett., 25 (1987), pp. 65–70.

[48] H. Rohnert, *A dynamization of the all-pairs least cost problem*, in Proc. 1985 Symposium on

Theoretical Aspects of Computer Science, Lecture Notes in Comput. Sci. 182, Springer-Verlag, Berlin, New York, Heidelberg, 1985, pp. 279–286.

[49] P. Rosenstiehl, personal communication.

[50] B. Schieber and U. Vishkin, *On finding lowest common ancestors: Simplification and parallelization*, SIAM J. Comput., 17 (1988), pp. 1253–1262.

[51] W. Schnyder, *Embedding planar graphs on the grid*, in Proc. ACM–SIAM Symposium on Discrete Algorithms, Society for Industrial and Applied Mathematics, Philadelphia, 1990, pp. 138–148.

[52] D. D. Sleator and R. E. Tarjan, *A data structure for dynamic trees*, J. Comput. System Sci., 24 (1983), pp. 362–381.

[53] R. Tamassia. *A dynamic data structure for planar graph embedding*, in Automata, Languages, and Programming (Proc. 15th International Colloquium on Automata, Languages, and Programming), Lecture Notes in Comput. Sci. 317, Springer-Verlag, Berlin, New York, Heidelberg, 1988, pp. 576–590.

[54] R. Tamassia and F. P. Preparata, *Dynamic maintenance of planar digraphs, with applications*, Algorithmica, 5 (1990), pp. 509–527.

[55] R. Tamassia and I. G. Tollis, *A unified approach to visibility representations of planar graphs*, Discrete Comput. Geom., 1 (1986), pp. 321–341.

[56] ———, *Dynamic reachability in planar digraphs with one source and one sink*, Theoret. Comput. Sci., 119 (1993), pp. 331–343.

[57] R. Tamassia and J. S. Vitter, *Parallel transitive closure and point location in planar structures*, SIAM J. Comput., 20 (1991), pp. 708–725.

[58] R. E. Tarjan and J. van Leeuwen, *Worst-case analysis of set-union algorithms*, J. Assoc. Comput. Mach., 31 (1984), pp. 245–281.

[59] R. E. Tarjan and A. C.-C. Yao, *Storing a sparse table*, Comm. Assoc. Comput. Mach., 22 (1979), pp. 606–611.

[60] J. Westbrook and R. E. Tarjan, *Maintaining bridge-connected and biconnected components on-line*, Algorithmica, 7 (1992), pp. 433–464.

[61] H. Whitney, *Non-separable and planar graphs*, Trans. Amer. Math. Soc., 34 (1932), pp. 339–362.

[62] M. Yannakakis, *Four pages are necessary and sufficient for planar graphs*, in Proc. 18th ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1986, pp. 104–108.

SIAM J. COMPUT.
Vol. 25, No. 5, pp. 998–1023, October 1996

PARALLEL SUFFIX–PREFIX-MATCHING ALGORITHM AND APPLICATIONS[*]

ZVI M. KEDEM[†], GAD M. LANDAU[‡], AND KRISHNA V. PALEM[§]

Abstract. Our main result in this paper is a parallel algorithm for *suffix–prefix- (s–p-) matching* that has optimal speedup on a concurrent-read/concurrent-write parallel random-access machine (CRCW PRAM). Given a string of length m, the algorithm runs in time $O(\log m)$ using $m/\log m$ processors. This algorithm is important because we utilize s–p matching as a fundamental building block to solve several pattern- and string-matching problems, such as the following: 1. string matching; 2. multitext/multipattern string matching; 3. multidimensional pattern matching; 4. pattern-occurrence detection; 5. on-line string matching. In particular, our techniques and algorithms are the first to preserve optimal speedup in the context of pattern matching in higher dimensions and are the only known ones to do so for dimensions $d > 2$.

Key words. amortized complexity, CRCW PRAMs, multidimensional pattern matching, parallel algorithms, pattern-matching automaton, speedup, string matching

AMS subject classifications. 68P99, 68Q10, 68Q22, 68Q25, 68Q68, 68R15, 68U15

1. Introduction.

Several important problems in computing involve the detection of repeated patterns within regular structures such as strings and higher-dimensional arrays. As a consequence, there is a rich history of fast algorithms for solving these problems. Karp, Miller, and Rosenberg [KMR72] used techniques based on successively refining equivalence classes of patterns of increasing size, where the patterns in an equivalence class are identical. Initially, they consider equivalence classes of patterns made up of a single character from the input. On each successive step, they construct equivalence classes of bigger pieces of the input by appropriately combining the equivalence classes from the previous step. Their algorithm was not optimal. Using a different approach, Weiner [W73] built a suffix tree and used it to design a linear-time algorithm for a fixed alphabet.

Two linear-time (optimal) algorithms for the *string-matching* problem were given by Knuth, Morris, and Pratt [KMP77] and Boyer and Moore [BM77]. These algorithms are based on the powerful notion of a *failure function.* Failure functions led to a substantial amount of subsequent work. Galil and Seiferas [GS83] have reported time- and space-optimal sequential, real-time algorithms for string matching. Aho and Corasick [AC75] have linear-time algorithms for a natural generalization of the string-matching algorithm in which the input has multiple patterns, possibly of different sizes.

Efficient algorithms for *multidimensional pattern matching* (or *d-dimensional pattern matching*) have been independently reported by Baker [Ba78], Bird [Bi77], and

[*] Received by the editors August 4, 1992; accepted for publication (in revised form) January 25, 1995. A preliminary version of this paper appeared in *Proc.* 1989 *ACM Symposium on Parallel Algorithms and Architectures.*

[†] Department of Computer Science, Courant Institute of Mathematical Sciences, New York University, 251 Mercer Street, New York, NY 10012-1185 (kedem@cs.nyu.edu). The research of this author was supported in part ONR contract N00014-85-K-0046 and NSF grant CCR-89-6949.

[‡] Department of Computer Science, Polytechnic University, 6 MetroTech, Brooklyn, NY 11201 (landau@poly.edu). The research of this author was partially supported by NSF grants CCR-8908286 and CCR-9305873 and the New York State Science and Technology Foundation, Center for Advanced Technology in Telecommunications, Polytechnic University, Brooklyn, NY.

[§] Department of Computer Science, Courant Institute of Mathematical Sciences, New York University, 251 Mercer Street, New York, NY 10012-1185 (palem@cs.nyu.edu).

Karp and Rabin [KR87] (randomized). These algorithms run in time[1] $O(d(n^d+m^d))$, given that the input text and pattern are of size n^d and m^d, respectively.

Parallel algorithms for string matching were given by Galil [G84] for strings from a fixed alphabet and Vishkin [V85] for strings from an arbitrary alphabet. Later, Breslauer and Galil [BG90], Vishkin [V91], and Galil [G92] designed new parallel algorithms for string matching with an arbitrary alphabet. Mathies [M88] and Amir and Landau [AL88] have presented parallel algorithms for solving the multidimensional pattern-matching problem. However, the techniques developed in these algorithms do not scale in the sense that they have not yielded optimal speedup when applied to pattern matching in higher dimensions. Our results in this paper are the first to solve pattern-matching problems in higher dimensions with optimal speedup. Subsequently and recently, Amir et. al [ABF93] and Cole et. al [CCG+93] presented techniques for achieving optimal speedup for the two-dimensional pattern-matching problem with an unbounded alphabet.

Our main result is an optimal-speedup parallel algorithm for solving the *suffix–prefix-matching (s–p-matching) problem.* Using this algorithm as the basic building block, we specify optimal-speedup parallel algorithms for several pattern- and string-matching problems; optimal-speedup parallel algorithms were not known for most of these problems before. (Given a problem instance of size n, we say that a parallel algorithm running in $T(n)$ steps using $P(n)$ processors performs *work* $P(n) \times T(n)$. We say that such an algorithm has *optimal speedup* if the work that it performs is (asymptotically) the same as the *running time* of the best-known *sequential* algorithm for solving the same problem.) The definition of the s–p-matching problem and a list of these other problems are given in §1.1. Our results have the advantage that they scale to higher dimensions while preserving optimal speedup.

In particular, our algorithm relies on the appropriate combination of two basic ideas. First, a novel aspect of our algorithm is that we construct a finite automation, as in Aho and Corasick [AC75], in parallel. This finite automaton can be used to recognize short strings of length $O(\log m)$, where m is the size of the input, in linear work. Since this step uses failure functions, we note that our algorithm is the first to use them in the parallel context. The second idea that we use involves computing *characteristics*, originally introduced in [AILSV88] and [KP92]. These characteristics are essentially the "short names" used by Karp, Miller, and Rosenberg [KMR72], as described in the first paragraph above. It is this combination that allows us to obtain optimal-speedup parallel algorithms for pattern matching in higher dimensions. All of our algorithms are developed in the context of an arbitrary concurrent-read/concurrent-write parallel random-access machine (CRCW PRAM) [J92].

The rest of this paper is organized as follows. In §1.1, the significant results in this paper are reported. Section 1.2 describes some previous work. Section 2 presents the new optimal-speedup algorithm for the s–p problem. Section 3 describes some applications of the s–p-matching algorithm. Section 4 gives some concluding remarks.

1.1. Significant results in this paper.

Our contributions are as follows:

1. We specify a parallel algorithm for the *s–p-matching problem* that has optimal speedup. The input to this problem consists of two equal-length strings A and B, both of length m. We wish to determine for each i, where $1 \le i \le m$, whether the

[1] In their paper, Karp and Rabin state that these algorithms ([Ba78], [Bi77], and [KR87]) run in time $O((n^d + m^d))$, but Karp, in personal communication, clarified that they assume a constant d.

suffix of A of size i is identical to the prefix of B of the same size. Our algorithm runs in $O(\log m)$ time using $m/\log m$ processors. The s–p-matching problem embodies a computational bottleneck in several pattern-matching problems. Therefore, using this algorithm as the fundamental building block, we are able to design the following optimal-speedup parallel algorithms.

2. We specify a new and simple algorithm that has optimal speedup for solving the string-matching problem with a polynomial-size alphabet. The input to this problem consists of two strings: a text of length n and a pattern of length m $(m \leq n)$. We wish to determine for each position of the text whether the pattern is equal to the substring of the text starting at it. Our algorithm runs in $O(\log m)$ time using $n/\log m$ processors. Except for this problem, optimal-speedup parallel algorithms were not known for the remaining four applications listed below.

3. We specify a new and simple algorithm that has optimal speedup for solving the *multitext/multipattern string-matching* problem. The input to this problem consists of u text strings $T_1, T_2, \dots, T_u$ of lengths $n_1, n_2, \dots, n_u$, respectively, and v patterns $P_1, P_2, \dots, P_v$, each of length $m \leq n_i$ for $1 \leq i \leq u$. We wish to determine for each position of each text string whether one of the patterns is equal to the substring of the text starting at it. Our algorithm runs in time $O(\log m)$ using $(vm + \sum_{j=1}^{u} n_j)/\log m$ processors.

4. We specify an optimal-speedup parallel algorithm for *multidimensional pattern matching*. The input to this problem consists of two arrays: a text of size n^d and a pattern of size m^d $(m \leq n, d \geq 1)$. We wish to determine for each position of the text whether the pattern is equal to the subarray of size m^d of the text starting at it. This algorithm runs in time $O(d \log m)$ using $n^d/\log m$ processors.

5. We specify an optimal-speedup parallel algorithm for solving the *occurrence-detection* problem. Informally, the input to this problem is a *text string* of size n presented as an (unordered) set of l *pieces* of size $k = n/l$, no two of them equal. The problem is as follows: given a pattern of size m, determine if there exists a concatenation of the l substrings to form a single string of size n such that the pattern occurs in the resulting string. This question is relevant to issues in molecular biology [CD88], [TU88]. Our algorithm solves this problem in time $O(\log m)$ using $n/\log m$ processors.

6. We specify a parallel algorithm for *on-line string matching* that has optimal *amortized* speedup. This is an on-line algorithm motivated by practical problems such as text editing. Here we have a text of size n and a pattern of size $m \leq n$ for which the string-matching problem has already been solved. Now, the text is extended by k characters, i.e., it is now of length $n + k$, and we wish to determine if there are any additional matches of the pattern in the text. Our algorithm solves this problem in time $O(\log m + \log k)$, and the *amortized work* (to be defined precisely later) done by it is always linear and hence optimal.

1.1.1. Remarks on alphabet size and space. Throughout the rest of this paper, we are concerned with an alphabet size that is at most polynomial in m. Given this, we assume without loss of generality that $\Sigma \subseteq \{0, 1, \dots, m-1\}$ and therefore the individual symbols of the alphabet (elements of Σ) are each $\log m$ bits long. Following standard conventions [J92], a processor is assumed to be able to read a constant number of symbols in a constant number of time units. We note that this is more general than the constant-sized alphabet used in [G84], for example. However, we do not consider alphabets of an arbitrary size relative to the size of the string, as in [V85]. All of our operations are standard PRAM operations. In contrast, in dealing with an arbitrary-sized alphabet (as in [V85]), it is assumed that any two symbols can be tested for equality in $O(1)$ time and work independent of the size. Given this assumption, we note that a problem with an arbitrary alphabet could be converted to one in which $|\Sigma| \leq m$ by sorting in $O(n \log m)$ work.

All of our techniques will work with space $O(m^{1+\epsilon})$ as in the work of Apostolico et al. [AILSV88] for any $0 < \epsilon \leq 1$ with a corresponding slowdown proportional to $1/\epsilon$. These implementations with reduced space requirements can be realized with just $O(m)$ cost (work) for initialization, following [H88].

1.2. Previous work. We now compare our applications of the s–p matching algorithm with previous results.

1. Galil [G84] and Vishkin [V85] have designed optimal-speedup parallel algorithms for string matching with input strings drawn, respectively, from a bounded and arbitrary alphabet. Both of these algorithms run in time $O(\log n)$ using $n/\log n$ processors of a CRCW PRAM. Breslauer and Galil [BG90], designed an algorithm that runs in time $O(\log\log n)$ using $n/(\log\log n)$ processors of a CRCW PRAM. Vishkin [V91] presented an algorithm whose text analysis runs in time $O(\log^* n)$ using $n/\log^* n$ processors of a CRCW PRAM, and Galil [G92] recently presented an algorithm that runs in constant time and uses a linear number of processors. This result has now been extended to the two-dimensional case [CCG+93] as well, using several new ideas.

2. Mathies [M88] presented a parallel algorithm for solving the multidimensional pattern-matching problem that runs in $O(d \log^2 n)$ time using n^d processors of a CRCW PRAM. Subsequently, Amir and Landau [AL88] presented an improved algorithm for the multidimensional pattern-matching problem that runs in $O(d \log m)$ time using n^d processors of a CRCW PRAM. Previous work does not yield an optimal speedup parallel algorithm for this problem.

2. An optimal-speedup algorithm for the s–p-matching problem.

In this section, we begin with a parallel algorithm for solving the s–p-matching problem that embodies some of the main ideas but does not have optimal speedup. Then, in the subsequent subsections, we progressively refine it by introducing additional techniques to eventually derive a parallel algorithm with optimal speedup in §2.4.

2.1. Computing characteristics and thereby deriving δ. In the interests of completeness, we start with a formal definition of the *s–p-matching problem*:

Input: Strings $A = a_0 a_1 \ldots a_{m-1}$ and $B = b_0 b_1 \ldots b_{m-1}$ over an alphabet Σ.

Output: A bit vector $\delta[0 \ldots m-1]$, where $\delta[i] = 1$, if and only if $a_{m-i-1} a_{m-i} \ldots a_{m-1} = b_0 b_1 \ldots b_i$.

The most straightforward computation of δ, which is used to solve the s–p-matching problem, would involve explicit manipulation of the m suffixes of A and the m prefixes of B. However, this is obviously inefficient. This inefficiency can be overcome by observing that the set S of these $2m$ suffixes and prefixes splits into at most $2m$ (and at least m) equivalence classes under the relation of equality. Therefore, there exists a function that maps S into the set $[1 \ldots 2m]$ with the property that two strings of the same length are mapped onto the same value if and only if they are equal. It will be sufficient for us to compute *any* such function in order to determine δ.

Therefore, throughout the rest of this section, we will be concerned with computing such a characteristic function χ (or *characteristic* for short) for a given set of strings. In particular, we wish to compute a characteristic function that maps the elements of such a set to "small" integers such that two elements of the set are mapped on the same integer if and only if they are equal. These χ values can then be used to quickly compute δ.

Without loss of generality, we will assume that m is a power of two. To help in the explanation, we will start with the assumption that we are given $c \times m$ processors for an appropriately chosen constant c. (Essentially, we will assume that there is a processor for every character in the given input.) Subsequently, we will extend this algorithm to one that only uses $m/\log m$ processors by using Brent's lemma [Br74]; the optimal-speedup result will follow from this extension.

2.2. A simple suboptimal algorithm. We first sketch a simple algorithm for computing δ in $O(m \log m)$ work and time $O(\log m)$. This algorithm computes the characteristics of the suffixes of A and the prefixes of B in S by appropriately combining the characteristics of their substrings.

Specifically, for each $i, i = 0, 1, \ldots, \log m$, we compute the characteristic of the set of all substrings of A and B of length 2^i, as was done in [AILSV88] and [KP92]. For the purpose of illustration, we will describe this computation for a value i assuming that the characteristics were computed for $i - 1$. We note that substrings of the same length are always handled concurrently. We first note that substrings of A and B are handled similarly here. Let $a_j a_{j+1} \ldots a_{j+2^i-1}$ be a substring of A. Dedicate a processor, say P_k $(0 \leq k \leq 2m - 1)$, to this substring. Given that the characteristics of strings of length 2^{i-1} were computed in the previous step, $\chi(a_j \ldots a_{j+2^{i-1}-1})$ and $\chi(a_{j+2^{i-1}} \ldots a_{j+2^i-1})$ are known and are in the range $[0 \ldots 2m - 1]$. This information is now combined to compute the characteristics of substring $a_j a_{j+1} \ldots a_{j+2^i-1}$ as follows. Processor P_k writes k into location number $\chi(a_j \ldots a_{j+2^{i-1}-1}) + 2m\chi(a_{j+2^{i-1}} \ldots a_{j+2^i-1})$ of some vector indexed by $0, 1, \ldots,$ $(2m)^2 - 1$. Then P_k reads the value in the above location and assigns this to $\chi(a_j, a_{j+1} \ldots a_{j+2^i-1})$. As all the processors write in parallel, only one of the processors writing into a location succeeds, and all processors writing into this location read that value. Moreover, the resulting χ value is in the range $[0 \ldots 2m - 1]$. Note that in phase 0 the vector is of size m, the size of the alphabet; processor P_k writes k into location a_k in the vector.

These characteristics are now combined appropriately to derive the χ values of the suffixes and prefixes in S. Assume that the empty string is assigned the characteristic of $2m$ (to make it different from those computed above). Let $\bar{a}$ be a suffix of A of some length $\sum_{i=0}^{\log m} c_i 2^i$, $c_i \in \{0, 1\}$. Write $\bar{a}$ as the concatenation $\bar{a}_{\log m}\bar{a}_{\log(m-1)} \cdots \bar{a}_0$, where the length of $\bar{a}_i$ is $c_i 2^i$ (that is, the length is 0 or 2^i). Do the same for each prefix $\bar{b}$ of B. In $\log m$ steps it is possible to compute characteristics for the set S by combining in step i, $i = 1, \ldots, \log m$, the values $\chi(\bar{a}_0 \ldots \bar{a}_{i-1})$ and $\chi(\bar{a}_i)$ to obtain $\chi(\bar{a}_0 \ldots \bar{a}_i)$. (To use the approach of the previous paragraph, replace $2m$ by $2m + 1$.) All $\bar{a}$'s and $\bar{b}$'s are processed in parallel as above to assure consistent assignment of characteristics to strings in S.

OBSERVATION 1. *The function δ can be computed in $O(m \log m)$ work and time* $O(\log m)$.

2.3. Characterizing prefixes efficiently. In this section, we refine the above algorithm to where the total work done is only $O(m)$ on the string B for which the characteristics of the *prefixes* have to be computed. This computation will run in time $O(\log m)$. However, the corresponding characterization of the *suffixes* of string A will still need $O(m \log m)$ work. We will return to a discussion of this issue in §2.3.2

below. Subsequently, in §2.4, we will refine these ideas further to get an algorithm that solves the s–p-matching problem in $O(\log m)$ time and $O(m)$ overall work.

In order to improve the work done in characterizing the prefixes of string B, we structure the computation to proceed in two stages, referred to respectively as the *winding* stage and the *unwinding* stage. The winding stage consists of phases $0, 1, \ldots, \log m$. In phase i, we compute the characteristic of the set of substrings $\{b_j \ldots b_{j+2^i-1} \mid 0 \leq j \leq m - 2^i$ and j divisible by $2^i\}$. Basically, in phase i, only those positions whose distance from the head of B is a multiple of 2^i are "active." The computation corresponding to all the other positions will have "gone to sleep." For each active position j, we compute the characteristic of the substring of length 2^i starting at j. Therefore, on phase i, we compute the characteristics of only $m/2^i$ substrings of B.[2]

In the (complementary) unwinding stage, the information computed in the winding stage is combined. Its schedule is the "reverse" of that of the winding stage, and its phases are $\log m - 2, \log m - 3, \ldots, 0$. In phase i, we compute the characteristics of the set $\{b_0 \ldots b_j \mid j > 2^i$, $j+1$ divisible by 2^i and not divisible by $2^{i+1}\}$. At this point, the characteristic of the prefix of B ending at position j is computed by combining previously computed characteristics of its substrings, as described in §2.2.

Remark. It is easily verified that the unwinding stage has two fewer phases than the winding stage. This is because the positions scheduled during phases $\log m$ and $\log m - 1$ in string B have their characteristics *completely* computed during the winding stage. Since only nodes whose characteristics are not completely computed during the winding stage need to be processed during the unwinding stage, we can drop the counterparts of these winding phases during unwinding. Therefore, we start with an unwinding phase of $\log m - 2$.

We will also introduce an example below to illustrate these issues.

2.3.1. An example. We will now present an example of the naming as it proceeds on the example strings A = *cabacaba* and B = *abacabab*. Essentially, in the algorithm described thus far, during the winding as well as the unwinding phases, the computation on A mimics the computation on B, so that the final χ values are correct. In other words, the algorithm must compute characteristics for suffixes of A that are *consistent* with those given to the prefixes of B. To do this, the intermediate steps in the computation of the characteristics of the suffix of A of any length j must follow the the computation of the characteristic of the prefix of B of length j.

In Table 1, we show the schedule based on which positions are active during the winding and the unwinding stages of string B. At each phase, we also show the characteristic values computed for the active positions.

In Table 2, we illustrate the way in which positions in string A are active and "keep up" with the computation on string B. The final value of each position is shown in boldface in Tables 1 and 2. As noted in the remark above, we note that the number of unwinding phases is two fewer than that of their winding counterparts. The results of the computations from Tables 1 and 2 are summarized below over all the phases

[2] The scheduling of computation on substrings of B is based on the algorithm from [KP92] used for computing characteristics of lineage functions of forests and is similar to the well-known prefix-sum computation [FL80]. Informally, we are given an input forest whose vertices and/or edges are labeled. A lineage function maps a set of labels of paths in this forest into some (range) set. However, since strings are a very special (degenerate) case of arbitrary forests, the techniques used here in the case of strings are significantly simpler than those used in the context of arbitrary forests for which the algorithm was originally designed.

TABLE 1
Execution trace on string $B = abacabab$.

		Winding Stage				Unwinding Stage	
Position	Processor Number	Phase 0	Phase 1	Phase 2	Phase 3	Phase 1	Phase 0
		χ	χ	χ	χ	χ	χ
0	8	**12**					
1	9	6	**15**				
2	10	12					**10**
3	11	4	11	**11**			
4	12	12					**12**
5	13	6	15			**13**	
6	14	12					**14**
7	15	6	15	15	**15**		

TABLE 2
Execution trace on string $A = cabacaba$.

		Winding Stage				Unwinding Stage	
Position	Processor Number	Phase 0	Phase 1	Phase 2	Phase 3	Phase 1	Phase 0
		χ	χ	χ	χ	χ	χ
0	0	4	0	4	**0**		
1	1	12	15	11		13	**14**
2	2	6	2	2		**2**	
3	3	12	11	3			**3**
4	4	4	0	**4**			
5	5	12	15				**10**
6	6	6	**2**				
7	7	**12**					

TABLE 3
Summary of characteristics of the substrings of A and B.

Phase 0: $a = 12$, $b = 6$, $c = 4$.
Phase 1: $ab = 15$, $ac = 11$, $ba = 2$, $ca = 0$.
Phase 2: $abac = 11$, $abab = 15$, $acab = 3$, $baca = 2$, $caba = 4$.
Phase 3: $abacabab = 15$, $cabacaba = 0$.
Phase 1: $abacab = 13$, $bacaba = 2$.
Phase 0: $aba = 10$, $abaca = 12$, $acaba = 3$, $abacaba = 14$.

in Table 3.

Let us consider a typical match of the prefix of B of size 7 with the suffix of A of the same length; the match is induced by the sequence $\sigma = abacaba$. Tracing through the tables, we see that these names are computed by decomposing σ into subcomputations on the three substrings as follows: $abac|ab|a$. Let us consider the unwinding stage in Tables 1 and 2 to understand this better. At the end of the winding stage, the characteristic of $abac$ is **11**, that of ab is **15**, and that of the last symbol a in isolation is **12**. In Phase 1 of the unwinding stage, processors 13 and 1 characterized the string $abacab$ to be **13**. Finally, in Phase 0, processors 1 and 14 computed the final characteristic of **14** in strings A and B, respectively, declaring the match.

It is easy to verify that the resulting characteristics of two substrings are always the same whenever they are identical. In the present implementation, we allow non-

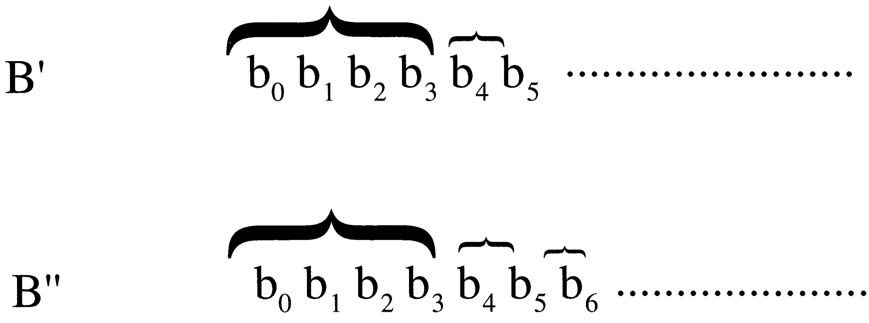

B'
b0 b1 b2 b3 b4 b5 ....
B''
b0 b1 b2 b3 b4 b5 b6 ....

FIG. 1. *Prefixes B' and B'' share intermediate substrings.*

identical strings of different lengths to sometimes get the same characteristic value. However, this does not cause any problem in solving the s–p-matching problem correctly.

2.3.2. The difficulty in characterizing suffixes efficiently. Recall that in the algorithm discussed above, during phase i of the winding stage, characteristics of substrings $\{a_j \dots a_{j+2^i-1} \mid 0 \leq j \leq m - 2^i\}$ of string A are computed. We observe that no more than $m - 2^i + 1$ substrings of A are active during this phase. Likewise, during phase i of the unwinding, for each k and j such that $0 \leq k \leq m - 2^i + 1$ and $j+1$ is *maximal* and is divisible by 2^i and not divisible by 2^{i+1}, the characteristics of the set $\{a_k \dots a_{k+j}\}$ of substrings of A are computed. For example, consider $m = 128$, $i = 5$. Then k is in the range of $0 \dots 97$. Now consider $k = 10$; we compute $j+1$ to be 96, and therefore the characteristics of $\{a_{10} \dots a_{105}\}$ will be computed. Once again, it is easy to verify that fewer than $m - 2^i + 1$ substrings of A are active during this phase.

Note that prefixes of B of lengths $\alpha 2^k + 1$ and $\alpha 2^k + 2$ for some α share many overlapping substrings. Indeed, it was this fact that allowed us in §2.3 to structure the winding and unwinding stages such that the only $O(m)$ work was done on string B overall. However, as shown in the example below, the "simulation" of this computation on the suffixes of A does not have this nice structure. In particular, suffixes of increasing lengths of string A do not share overlapping substrings in such a simulation. As such, it is not hard to verify that by the end of the winding stage, we would have computed the characteristics of a set of $O(m \log m)$ substrings of A but of only $O(m)$ substrings of B. We will now sketch a brief example to better illustrate this difficulty. In Figure 1, we have an example string B and we consider two prefixes of it of six characters and seven characters each and denoted by B' and B'', respectively. Similarly, we consider an example string A shown in Figure 2, and two of its suffixes in turn also with six and seven characters are A' and A'', respectively.

Clearly, one possible case of s–p matching is where the suffix A' of A of length six is aligned with the prefix B' of B of the same length. Similarly, the second case is where A'' is aligned with B''. As shown in the figures (and explained above), prefixes B' and B'' share intermediate substrings that are composed during the naming process. For example, the characteristic of B'' is derived simply by adding the single character b_6 to the characteristic of B'. However, as we can see from Figure 2, this is not true of the corresponding suffix A'' with respect to A'.

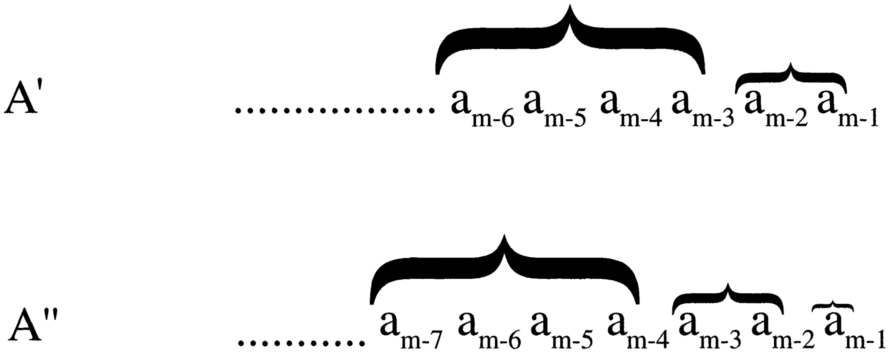

A'
.... $a_{m-6}\,a_{m-5}\,a_{m-4}\,a_{m-3}\,a_{m-2}\,a_{m-1}$
A''
.... $a_{m-7}\,a_{m-6}\,a_{m-5}\,a_{m-4}\,a_{m-3}\,a_{m-2}\,a_{m-1}$

FIG. 2. *Suffixes A' and A'' do not share intermediate substrings.*

2.4. The optimal algorithm. Recall from the previous section that A was the "difficult" string as it required $O(m \log m)$ work. Intuitively, the way in which we get around this is to shrink A to a string of size $m/\log m$. This is done by first partitioning it into $m/\log m$ nonoverlapping substrings, each of length $\log m$. We then replace each substring by its characteristic value χ to get a new string $\bar{A}$. In conjunction with this, we decompose B into $\log m$ strings, $\bar{B}_1, \bar{B}_2, \ldots \bar{B}_{\log m}$, each of length $m/\log m$, as follows: character i in the jth such string characterizes the $\log m$-length substring of B starting at position $j + i \log m$. We operate on each of these $\log m$ cases generated by B independently using the single copy of $\bar{A}$ as described in §2.3.

The algorithm is stated concisely in §2.4.1. This is followed by a detailed example in §2.4.2 that illustrates this concise description. In order to achieve parallel speedup, the algorithm for s–p matching discussed here relies on a parallel construction of the well-known Aho–Corasick automation from [AC75]. We describe the details of this parallel construction in §2.4.3. This construction is used to implement steps 1 and 2 from §2.4.1, as shown in §2.4.4. Subsequently, step 3 of the algorithm is described in §§2.4.5 and 2.4.6. The details of coping with "boundary conditions" in step 4 are discussed in §2.4.7, and solving the s–p-matching problem in step 5 is summarized in §2.4.8. Finally, the complexity of the overall algorithm is analyzed in §2.4.9.

2.4.1. Concise statement of the algorithm.

1. Compute in parallel the Aho–Corasick automaton M representing the set of $m/\log m$ nonoverlapping substrings of A of length $\log m$ each:

$$\begin{array}{l} a_0 a_1 \ldots a_{\log m-1}, \\ a_{\log m} a_{\log m+1} \ldots a_{2\log m-1}, \\ \vdots \\ a_{m-\log m} a_{m-\log m+1} \ldots a_{m-1}. \end{array}$$

Let $\bar{a}_i$ for $i = 0, \log m, \ldots, m - \log m$ be the name (number) of the state *accepting* the substring $a_i a_{i+1} \ldots a_{i+\log m-1}$. (We note that a state $\bar{a}_i$ for $i = 0, \log m, \ldots, m - \log m$ accepts the substring $a_i a_{i+1} \ldots a_{i+\log m-1}$ if and only if from the start state, the sequence of transitions induced by $a_i a_{i+1} \ldots a_{i+\log m-1}$ lead to state $\bar{a}_i$.) Create the string $\bar{A} = \bar{a}_0 \bar{a}_{\log m} \bar{a}_{2\log m} \ldots \bar{a}_{m-\log m}$.

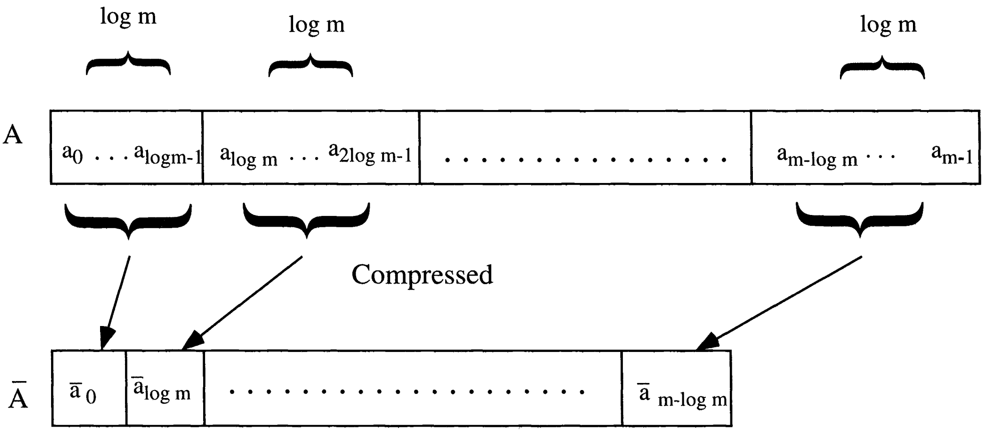


FIG. 3. *The transformation done to string A.*

2. Apply in parallel the automaton M to the string B considered as text. As the result, create the string $\bar{B} = \bar{b}_0\bar{b}_1 \ldots \bar{b}_{m-\log m}$, where $\bar{b}_i$ is the state accepting the string $b_i b_{i+1} \ldots b_{i+\log m-1}$.

Create $\log m$ strings

$$\begin{aligned}\bar{B}_1 &= \bar{b}_1\bar{b}_{\log m+1}\bar{b}_{2\log m+1}\cdots\bar{b}_{m-2\log m+1},\\ \bar{B}_2 &= \bar{b}_2\bar{b}_{\log m+2}\bar{b}_{2\log m+2}\cdots\bar{b}_{m-2\log m+2},\\ &\vdots\\ \bar{B}_{\log m} &= \bar{b}_{\log m}\bar{b}_{2\log m}\cdots\bar{b}_{m-\log m}.\end{aligned}$$

At this point, we have "compressed" string A by a factor of $\log m$, as shown in Figure 3.

Furthermore, we have also "decomposed" string B into $\log m$ components, each of length $m/\log m$; one such decomposed component is illustrated in Figure 4.

3. Compute in parallel the characteristics of the set consisting of all the proper suffixes of the string $\bar{A}$ and all the proper prefixes of the strings $\bar{B}_1, \bar{B}_2, \ldots, \bar{B}_{\log m}$.

4. Compute in parallel the characteristics of the set consisting of all the suffixes of the set of strings

$$\begin{aligned}&a_0a_1\ldots a_{\log m-1},\\ &a_{\log m}a_{\log m+1}\cdots a_{2\log m-1},\cdots,\\ &a_{m-\log m}a_{m-\log m+1}\cdots a_{m-1}\end{aligned}$$

and all the prefixes of the string $b_0b_1\ldots b_{\log m-1}$.

This part of the computation is used in processing the "remainder" pieces as shown in Figure 4.

5. Compute in parallel the vector δ. Let j, where $1 \leq j \leq m$, be such that $j = c_1\log m + c_2$, where $c_1 \geq 0$ and $1 \leq c_2 \leq \log m$. Also, let $i = j-1$. Then $\delta[i] = 1$ if and only if

(a) $\chi(\bar{a}_{m-c_1\log m}\bar{a}_{m-(c_1-1)\log m}\cdots\bar{a}_{m-\log m}) = \chi(\bar{b}_{c_2}\bar{b}_{c_2+\log m}\cdots\bar{b}_{c_2+(c_1-1)\log m})$ and

(b) $\chi(a_{m-i-1}a_{m-i}\ldots a_{m-c_1\log m-1}) = \chi(b_0b_1\ldots b_{c_2-1})$.

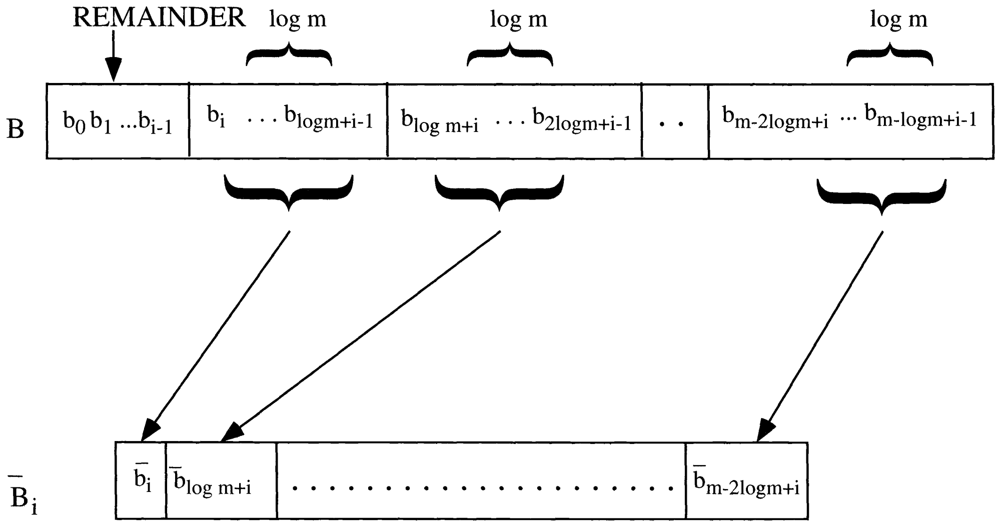


FIG. 4. *The manner in which string B decomposes into $\bar{B}_i$.*

2.4.2. Example. We now present an example. Let $\Sigma = \{a, b\}$, $m = 16$, and

$$A = bbabbbaaabaabbab, \qquad B = aabaabbabababaaa.$$

1. Decompose $A = bbab|bbaa|abaa|bbab$ (we broke A into strings of length $\log_2 16$ = 4 each). We now need to construct an automaton representing the four (actually three distinct) substrings. The automaton has 10 states. Its starting state is ϕ, and it is defined by the functions listed in Table 4 below and represented in Figure 5. In this figure, the "goto" function is denoted by the solid lines, whereas the "failure" function is indicated by the dashed or broken lines; please refer to §2.4.3 for a detailed review of the structure of such an automaton.

TABLE 4.

state	symbol	g(state, symbol)
ϕ	a	α
ϕ	b	ϵ
α	a	
α	b	β
β	a	γ
β	b	
γ	a	δ
γ	b	
δ	a	
δ	b	
ϵ	a	
ϵ	b	ζ
ζ	a	η
ζ	b	
η	a	θ
η	b	ι
θ	a	
θ	b	
ι	a	
ι	b	

TABLE 4 (*cont.*).

state	f(state)
ϕ	ϕ
α	ϕ
β	ϵ
γ	α
δ	α
ϵ	ϕ
ζ	ϵ
η	α
θ	α
ι	β

δ stands for *abaa*, θ for *bbaa*, ι for *bbab*. Thus $\bar{A} = \iota\theta\delta\iota$,
2.

$$\bar{B} = \gamma\delta\beta\zeta\eta\iota\gamma\beta\gamma\beta\gamma\delta\alpha,$$

$$\bar{B}_1 = \delta\iota\beta, \qquad \bar{B}_2 = \beta\gamma\gamma, \qquad \bar{B}_3 = \zeta\beta\delta, \qquad \bar{B}_4 = \eta\gamma\alpha.$$

3. The relevant (distinct) substrings consisting of certain suffixes of $\bar{A}$ and prefixes of $\bar{B}_1, \bar{B}_2, \bar{B}_3$, and $\bar{B}_4$ are β, δ, ζ, η, ι, $\beta\gamma$, $\delta\iota$, $\zeta\beta$, $\eta\gamma$, $\beta\gamma\gamma$, $\delta\iota\beta$, $\zeta\beta\delta$, $\eta\gamma\alpha$, and $\theta\delta\iota$, and we may assume that characteristics have been computed for them appropriately.

4. The relevant (distinct) substrings are *a*, *b*, *aa*, *ab*, *aab*, *baa*, *bab*, *aaba*, *abaa*, *bbaa*, and *bbab*, and we may assume that characteristics have been computed for them appropriately.

5. Let us compute $\delta[8]$. $i = 8$, $j = i + 1 = 9$. Thus $j = 2 \cdot 4 + 1$, and therefore $c_1 = 2$ and $c_2 = 1$. Note that $\bar{a}_8\bar{a}_{12} = \bar{b}_1\bar{b}_5 = \delta\iota$; $\delta\iota$ stands for *abaabbab*. The equality of $\bar{a}_8\bar{a}_{12} = \bar{b}_1\bar{b}_5$ is determined by checking the characteristics of $\bar{a}_8\bar{a}_{12}$ and $\bar{b}_1\bar{b}_5$. Furthermore, $a_7 = b_0 = a$, which is also checked by characteristics. Thus $\delta[8] = 1$. Indeed, the suffix of length 9 of A and the prefix of length 9 of B are both equal to *aabaabbab*.

Before proceeding any further, we will first describe the construction of the Aho–Corasick automaton in detail. It has to be recalled that this construction is done in step 1 of the algorithm described in §2.4.1.

2.4.3. The Aho–Corasick automaton and its parallel construction. We assume that the reader is to some extent familiar with the work of Aho and Corasick from [AC75]. However, we will briefly review that work and fix the notation now. First, given several pattern strings, a tree (trie) describing them is constructed. This tree describes the underlying automaton (see Figure 5) whose states correspond to the nodes. We have a "goto" function g(state $= r$, symbol $= a$) represented by the solid lines in Figure 5 that indicates the next state to go to if we are in state r and the current symbol in the text string being read is a. In other words, $g(r, a) = s$ if the prefix can be extended by the symbol a. The new state will be s. Let $\eta(r)$ denote the *depth* of the state r, that is, the distance between it and the root (ϕ in Figure 5) of the tree. The automaton is in state r if and only if the following conditions hold:

(i) The suffix of the prefix of the text string examined so far is equal to the labels of the states from the root to r. Let α denote this sequence of labels to r.

(ii) Furthermore, α is equal to a prefix of some pattern string(s).

(iii) Also, $\eta(r)$ is the largest possible match found thus far ending in the text-string position being currently matched.

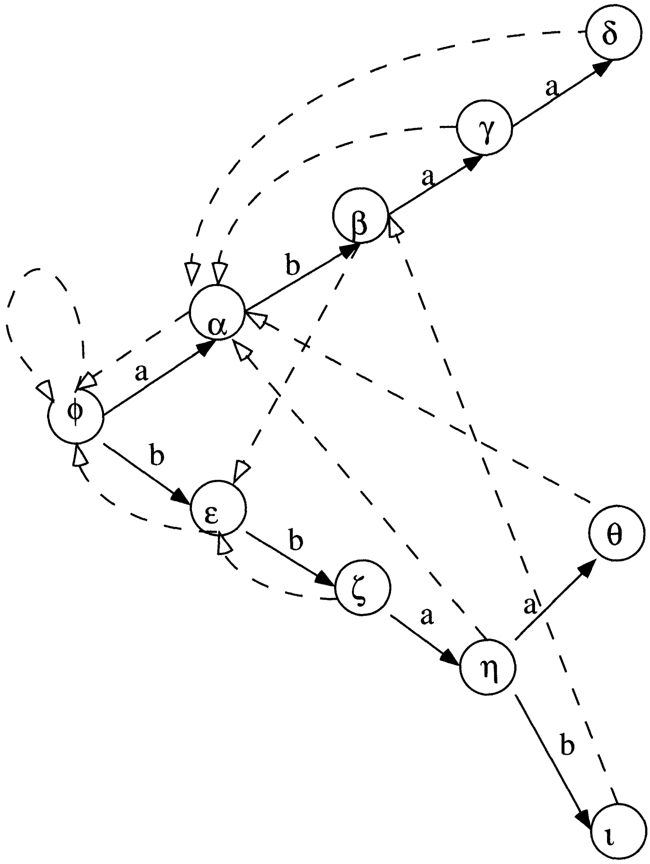


FIG. 5. *The Aho–Corasick automation for this example.*

We also have the "failure" function $f(r)$ represented by the broken lines in Figure 5. The failure function is used to continue matching the pattern on the string should $g(r, a) =$ fail. This indicates that there is no pattern string whose prefix is $\alpha.a$, where . denotes concatenation as a prefix. Equivalently, any symbol of the text being read cannot be used to extend the prefix. That is, if $g(r, a)$ for the symbol a that was read is undefined, the automaton goes to state $f(r)$ and processes a again. In other words, it checks whether $g(f(r), a)$ is defined. If it is, the transition takes place; otherwise, $f(f(r))$ is checked, and so on.

2.4.3.1. The parallel construction. We start by assigning a processor to each pattern string. Recall that, conceptually, each state of the automaton corresponds to a substring of one or more of the pattern strings. Informally, each state of the automaton is associated with an array of size $|\Sigma|$.[3] Let us suppose that the automaton has been constructed to include all states of depth d or less. Let r be a state at depth d. Then, if $g(r, a) = s$, the parallel algorithm first allocates a new array of size $|\Sigma|$

[3] Recall that in our paper, we are concerned with a $|\Sigma|$ that is polynomially bounded in the size of the input strings. Therefore, the automaton can be trivially implemented using a polynomial amount of space.

to represent s. This is done by (one of) the processor(s) allocated to a pattern string whose prefix of length $d+1$ defined state s. Following this step, this processor adds a pointer in the position corresponding to a (in the array representing r) to point to the array representing s. This pointer implements $g(r, a) = s$.

We will now address the question of computing the failure function f. However, before proceeding with this parallel construction, it is helpful to briefly review its classical sequential construction. The sequential algorithm computes f iteratively based on the depth d of the state s. Let r be the (unique) parent state of s in the corresponding tree representation. In particular, let $g(r, a) = s$. Informally, the iterative process involves following the failure functions towards the root of the trie until a state s' is encountered such that

1. the string of symbols representing the path from the root to $\sigma' = s'$ is a proper suffix of the sequence of symbols encoding the path from the root to state s and
2. σ' is the longest sequence with the above property.

Equivalently, let $\eta(r) = d - 1$ and let $g(r, a) = s$. Let $r_1, r_2, \ldots, r_\nu$ ($\nu \geq 1$) be the shortest sequence with the following properties: $r_1 = r$, $r_{i+1} = f(r_i)$ for $i \geq 1$ and $g(r_\nu, a)$ is defined. Then $f(s) = g(r_\nu, a)$.

Returning to the question of constructing the failure functions in parallel, we will now show that the parallel algorithm will proceed in at most $O(D)$ steps, where D denotes the length of the longest pattern string in the input.[4] For any state x, we denote by $\tau(x)$ the step in which the computation of $f(x)$ is finished. We will show the following.

THEOREM 2.1. *For any state s, $\tau(s) \leq 2\eta(s) - 1 - \eta(f(s))$.*

Proof. Again, let r, s, and a be such that $g(r, a) = s$. The computation of $f(s)$ will start in the step following the step in which $f(r)$ was computed. By (informal) induction on the depth of a state, $f(r)$ is computed by step $2\eta(r) - 1 - \eta(f(r))$. We will show that it will end no later than in step $2\eta(s) - 1 - \eta(f(s))$.

During the computation, the processor P allocated to s (technically, this is one of the processors allocated to the string whose prefix of length $\eta(s)$ terminates at s) follows the sequence $f(r_1), f(r_2), \ldots, f(r_{\nu-1})$, $\nu \geq 1$, described above, with one step required for examining each f (and associated g). However, we need to ensure that processor P does not wait by more than a constant amount of time to determine each of the $f(r_i)$ for $1 \leq i \leq (\nu - 1)$, or else processor P might have to "wait" for the processor computing $f(r_i)$ to complete its computation.

We will therefore first prove the following crucial bound on the time that relates $\tau(r_i)$ and $\tau(s)$. Specifically we have the following.

LEMMA 2.2. *For $1 \leq i \leq (\nu - 1)$, $\tau(r_i) \leq 2\eta(s) - \eta(f(s)) - \nu - i + 1$.*

Proof. We first recall that $f(r_i) = r_{i+1}$ and that $\eta(r_i) > \eta(r_{i+1})$ by construction. Therefore, we note that $\eta(s), \eta(r_1), \eta(r_2), \ldots, \eta(r_{\nu-1})$ is a strictly decreasing sequence of integers. From the above, it immediately follows that

$$\eta(r_i) \leq \eta(s) - i. \tag{1}$$

From the monotonicity of the depths of the sequence r_i, it also follows that

$$\eta(r_\nu) \leq \eta(r_i) - (\nu - i) \tag{2}$$

[4] In our actual application, all the pattern strings are of (equal) $D = \log m$ length.

or, since $\eta(f(s)) = \eta(r_\nu) + 1$,

$$\eta(f(s)) - 1 \leq \eta(r_i) - (\nu - i). \tag{3}$$

Equivalently, replacing i with $i+1$, we have

$$\eta(f(s)) - 1 \leq \eta(r_{i+1}) - (\nu - i - 1). \tag{4}$$

From (4) above and the fact that $\eta(f(r_i)) = \eta(r_{i+1})$, it immediately follows that

$$\eta(f(s)) + (\nu - i) - 2 \leq \eta(f(r_i)). \tag{5}$$

By our induction on the depth of the state, recall that

$$\tau(r_i) \leq 2\eta(r_i) - 1 - \eta(f(r_i)) \tag{6}$$

for $i < \nu$. Substituting in (6) for $\eta(r_i)$ and $\eta(f(r_i))$ from (1) and (5) above, respectively, and simplifying, we have

$$\tau(r_i) \leq 2\eta(s) - \eta(f(s)) - \nu - i + 1. \tag{7}$$

From the lemma above, we deduce that $f(r_1)$ is the last of the failure functions to be computed from the sequence $f(r_1), f(r_2) \ldots f(r_{\nu-1})$. Furthermore, it is computed by $T = 2\eta(s) - \eta(f(s)) - \nu$, derived by substituting $i = 1$. This implies that all the required $\nu - 1$ values of f are known by step T. Let us now recall that in order to compute $f(s)$, processor P follows the sequence $f(r_1), f(r_2), \ldots, f(r_{\nu-1})$, $\nu \geq 1$, described above, with one step required for examining each f (and associated g). Assuming that processor P starts the computation of $f(s)$ after time step T we conclude that this computation is completed by step $T + (\nu - 1) = 2\eta(s) - \eta(f(s)) - 1$, and the theorem is proved. □

2.4.4. Computing characteristics of substrings of length log m. From the construction in §2.4.3, it is easy to compute the characteristics of the set of $m/\log m$ substrings $\bar{a}_0, \bar{a}_{\log m}, \ldots, \bar{a}_{m-\log m}$ of A (used to derive $\bar{A}$). Since these strings are all of equal (short) length of $\log m$ symbols, we have the following from Theorem 2.1.

OBSERVATION 2. *The Aho–Corasick automaton M that accepts strings $\bar{a}_0, \bar{a}_{\log m}, \ldots, \bar{a}_{m-\log m}$ can be constructed in work $O(m)$ and time $O(\log m)$. The χ values for the substrings of A are simply the names of the states that accept them in the automaton.*

We now proceed to assign characteristics to the $m - \log m$ substrings of B as follows. Dedicate a processor P_k (here $0 \leq k \leq m/\log m - 1$) to compute the characteristics $\bar{b}_{k \log m}, \bar{b}_{k \log m+1}, \ldots, \bar{b}_{(k+1)\log m-1}$ of the $\log m$ substrings

$$b_{k\log m} \ldots b_{(k+1)\log m-1}, b_{k\log m+1} \ldots b_{(k+1)\log m}, \ldots, b_{(k+1)\log m-1} \ldots b_{(k+2)\log m-2}.$$

This is easily done by running the automaton on the string $b_{k \log m} \ldots b_{(k+2)\log m-2}$ sequentially! If a substring of length $\log m$ is accepted, we characterize it by the name of the accepting state. If it is not accepted, we characterize it by the name of the state the automaton reached after processing it. This sequential computation is done in $O(\log m)$ time per processor. (Note that we are not actually assigning distinct characteristics to the substrings of B that are not equal to any of the $m/\log m$ substrings of A. As it turns out, this does not affect the correctness of our algorithm.)

2.4.5. A sketch of one of the cases. Assume for now that we have computed the characteristics of the following set of $2m/\log m$ substrings of length $\log m$ of A and B: $\bar{a}_i = \chi(a_i \dots a_{i+\log m-1})$ and $\bar{b}_i = \chi(b_i \dots b_{i+\log m-1})$, where i is a positive-integer multiple of $\log m$. $\bar{a}_i$ and $\bar{b}_i$ are symbols from a suitable alphabet and can be encoded in $O(\log m)$ bits. Now consider the two strings $\bar{A} = \bar{a}_0\bar{a}_{\log m}\ \bar{a}_{2\log m} \dots \bar{a}_{m-\log m}$ and $\bar{B} = \bar{b}_0\bar{b}_{\log m}\ \bar{b}_{2\log m} \dots \bar{b}_{m-\log m}$. These two strings are of length $m/\log m$ each. The characteristics of the suffixes and the prefixes defined by these two strings can therefore be computed in work $O((m/\log m)\log(m/\log m)) = O(m)$ using our algorithm from §2.3. This will give a *partial* solution to the problem for the original input strings A and B. Specifically, we solve the problem of computing the characteristics, but only for the suffixes of A and the prefixes of B whose lengths are divisible by $\log m$. Notice that in this computation, $O(m)$ work was devoted to $\bar{A}$ derived from A, but only $O(m/\log m)$ work was devoted to $\bar{B}$ derived from B. The extension of the approach to handle the *complete* problem will balance the requirements so that $O(m)$ work is devoted to both strings derived by A and those derived from B as well.

2.4.6. Computing characteristics for the original inputs. To generalize from §2.4.5, assume that we have computed the characteristics of the following substrings of A and B, each of length $\log m$: $b_i b_{i+1} \dots b_{i+\log m-1}$ for $i = 0, 1, \dots, m - \log m$ and $a_i a_{i+1} \dots a_{i+\log m-1}$ for i divisible by $\log m$. Denote

$$\bar{b}_i = \chi(b_i b_{i+1} \dots b_{i+\log m-1}),$$

$$\bar{a}_i = \chi(a_i a_{i+1} \dots a_{i+\log m-1}).$$

We will now proceed to solve the original problem by considering suffixes of the single string $\bar{A}$ and prefixes of the $\log m$ strings derived from B of the form

$$\begin{aligned}
\bar{B}_1 &= \bar{b}_1\bar{b}_{\log m+1}\bar{b}_{2\log m+1} \dots \bar{b}_{m-2\log m+1},\\
\bar{B}_2 &= \bar{b}_2\bar{b}_{\log m+2}\bar{b}_{2\log m+2} \dots \bar{b}_{m-2\log m+2},\\
&\vdots\\
\bar{B}_{\log m} &= \bar{b}_{\log m}\bar{b}_{2\log m} \dots \bar{b}_{m-\log m}.
\end{aligned}$$

It can be verified that, disregarding "remainder" strings of length $\le \log m$ (this is sketched in §2.4.7), every potential matching of a suffix in A with a prefix of B is covered by one of these $\log m$ strings. Therefore, we characterize the set of all suffixes of $\bar{A}$ and the prefixes of each of the $\log m$ strings $\bar{B}_1, \bar{B}_2, \dots, \bar{B}_{\log m}$. Furthermore, from the previous discussion, the next observation follows immediately.

OBSERVATION 3. *The characteristics of $\bar{A}$ can be computed in $O(m)$ and those of each of the $\bar{B}_i$ in work that is linear in the size of $(O(m/\log m))$. Hence the total work done is $O(m)$.*

2.4.7. Computing characteristics of the remainders. In B, there are only $\log m$ distinct "remainder" strings of the form $b_0 b_1 \dots b_x$, where $0 \le x \le \log m - 1$. In other words, these are all possible prefixes of the substring formed by the first $\log m$ positions of B. In A, there are a total of m possible remainders. To see this, let us consider A as being decomposed into $m/\log m$ disjoint substrings each of length $\log m$. Each of these substrings contributes exactly $\log m$ of its suffixes as possible remainders. For example, consider the substring $a_{c\log m} \dots a_{(c+1)\log m-1}$. In this case, the remainders are all of its suffixes including the substring itself. This computation can be viewed as solving $m/\log m$ s–p-matching problems, one for each of the substrings of A and a unique pattern string from B. However, the pattern in this case is only $O(\log m)$ characters long. Therefore, to solve the problem in parallel, we need to assign only $\log^2 m$ processors; $\log m$ proccessors are assigned to each of the distinct prefixes of B. Consider one of the substrings of A in question of length $\log m$. With a single processor, its suffixes are named using the techniques outlined in §2.3 in $O(\log m)$ time (and work). (Note that in §2.3, the computation of the prefixes is efficient while the computation of the suffixes is less efficient. One can easily reverse this, namely, design an algorithm that has linear work on the suffix computation and $O(m \log m)$ work on the prefix computation.) Therefore, we have the following.

OBSERVATION 4. *The characteristics of the set of the remainder strings can be computed in $O(m)$ work and $O(\log m)$ time using up to $m/\log m$ processors.*

2.4.8. s–p matching from characteristics.

We use characteristics to compute the vector δ.

Let $j \in \{1, 2, \ldots, m\}$, and we write $j = c_1 \log m + c_2$, where $c_1 \geq 0$ and $1 \leq c_2 \leq \log m$. Also, let $i = j - 1$. Then $\delta[i] = 1$ if and only if

1. $a_{m-c_1 \log m} a_{m-c_1 \log m+1} \cdots a_{m-1} = b_{c_2} b_{c_2+1} \cdots b_{i=c_2-1+c_1 \log m}$ and
2. $a_{m-i-1} a_{m-i} \cdots a_{m-c_1 \log m-1} = b_0 b_1 \ldots b_{c_2-1}$.

It is easy to see that the first condition is equivalent to verifying the following relationship between characteristics:

$$\chi(\bar{a}_{m-c_1 \log m} \bar{a}_{m-(c_1-1) \log m} \cdots \bar{a}_{m-\log m}) = \chi(\bar{b}_{c_2} \bar{b}_{c_2+\log m} \cdots \bar{b}_{c_2+(c_1-1) \log m}).$$

This condition checks whether the suffix of length c_1 of $\bar{A}$ (of length $c_1 \log m$ in A) is equal to the prefix of length c_1 of $\bar{B}_{c_2}$ (substring of length $c_1 \log m$ of B that starts with b_{c_2}). The second condition checks whether the remainders of length c_2 in both A and B are equal.

2.4.9. Complexity.

We first state the time/processor complexity of the algorithm. The proof of the theorem is not given here as it follows in a straightforward manner from the statement of the algorithm and the discussion and observations in the previous sections.

THEOREM 2.3. *The above algorithm solves the s–p-matching problem in $O(\log m)$ time using work of $O(m)$ on a CRCW PRAM given input strings A and B of size m each.*

Given this theorem and using Brent's lemma, we immediately obtain an algorithm that solves the s–p-matching problem in $O(\log m)$ time using $m/\log m$ processors (of a CRCW PRAM as well) given input strings A and B of size m each. Beginning with §3, we will use this modified algorithm.

To reiterate, our algorithm relies on two basic ideas. These ideas are

1. computation of characteristics and
2. the usage of failure functions for recognizing very short strings of equal length by means of the Aho–Corasick automaton.

Using the first idea alone will give us a suboptimal-speedup algorithm of work $O(m \log \log m)$. This is because we can replace the Aho–Corasick automaton and its application in the naming of the strings in steps 1 and 2 of our algorithm (§2.4.1) with a procedure for computing characteristics. It is a simple exercise to verify that this replacement will require an additional $O(\log \log m)$ multiplicative work overhead due to the computation on B.

3. Applying the s–p matching algorithm.

We will now describe various applications of our algorithm for s–p matching.

3.1. Multiple s–p-matching problems. Most of the subsequent algorithms can be put in a setting of simultaneously solving the matching of several suffixes against several prefixes. Say we are given u text strings $T_1, T_2, \ldots, T_u$ of lengths $n_1, n_2, \ldots, n_u$, respectively, and v patterns $P_1, P_2, \ldots, P_v$, each of length $m \leq n_i$ for $1 \leq i \leq u$. We wish to determine which suffixes of the T_i's match which prefixes of the P_j's. It is possible to formulate an algorithm for this general problem at this point, but instead we will develop various special cases of it as needed.

3.2. String matching. The classical string-matching problem is defined as follows:

Input: a *pattern* string of length m and a *text* string of length $n \geq m$.

Output: all the positions in the text in which the pattern matches.

We will show how to reduce the solution of the problem to the solution of a version of the multiple s–p-matching problem.

Let the pattern string be $P = p_1 p_2 \ldots p_m$ and let the text string be $T = t_1 t_2 \ldots t_n$. "Cut" T into $\lceil n/m \rceil$ nonoverlapping substrings $T_1, T_2, \ldots, T_{\lceil n/m \rceil}$.

$$T_j = t_{(j-1)m+1} t_{(j-1)m+2} \cdots t_{jm}$$

for $j < \lceil n/m \rceil$ and $T_{\lceil n/m \rceil} = t_{\lceil n/m \rceil} t_{\lceil n/m \rceil + 1} \cdots t_n$. Thus $T = T_1 T_2 \ldots T_{\lceil n/m \rceil}$.

It is easy to see that the pattern matches some position in the text if and only if for some j, a suffix of length $k > 0$ of T_j matches a prefix of P and a prefix of length $m - k$ of T_{j+1} matches a suffix of P. If $k = m$ or $j = \lceil n/m \rceil$, then $j + 1$ is undefined. This observation can be used to immediately produce an algorithm for string matching. For a simple example, consider $P = abaa$, $T = babaaaaabaa$. Here $T_1 = baba$, $T_2 = aaaa$, and $T_3 = baa$.

P matches T in position 2 because of the following reasons:

1. aba is both a suffix of T_1 and a prefix of P.
2. a is both a prefix of T_2 and a suffix of P.

Thus, in general, we need to solve two subproblems:

1. the s–p-matching problems for finding the matches between all the suffixes of the strings $T_1, T_2, \ldots, T_{\lceil n/m \rceil}$ and the prefixes of the string P;
2. the s–p-matching problems for finding the matches between all the suffixes of the string P and the prefixes of the strings $T_1, T_2, \ldots, T_{\lceil n/m \rceil}$.

These two subproblems can be solved in time $O(\log m)$ and work $O(n)$ by a straightforward application of the standard s–p algorithm. For instance, the first subproblem can be solved by *parallel* solution of the $\lceil n/m \rceil$ s–p problems, each defined by a pair of strings (T_j, P). This will characterize the set consisting of all the suffixes of $T_1, T_2, \ldots, T_{\lceil n/m \rceil}$ and the prefixes of P. ($T_{\lceil n/m \rceil}$ could, in general, be shorter than m, but this is not significant.) Again, following Brent's lemma we have the result below.

THEOREM 3.1. *Given a text string of length n and a pattern of length m, the algorithm above solves the string matching problem using $n/\log m$ processors in $O(\log m)$ time of a CRCW PRAM.*

3.3. Multipattern string matching. We will now define the multipattern string-matching problem.

Input: a *text* string of length n and v *patterns* each of length m. (The patterns are not necessarily distinct.)

Output: for each position in the text, indicate if a pattern matches there. If a match exists, report one of the matching patterns.

Let T be the text string, and let $P_1, P_2, \ldots P_v$ be the patterns. Again consider an example first. Let

$$T = abbbab,$$
$$P_1 = P_2 = ab, \qquad P_3 = ba.$$

Then ab matches T at positions 1 and 5, and ba matches T at position 4. Thus there are four "acceptable" representations of the output depending on whether we state that P_1 or P_2 match at positions 1 or at position 5. Thus if 0 indicates no match, the answer could be given by any of the following four strings: $10031, 10032, 20031, 20032$. To make the presentation easier, in the rest of the paper, we will represent outputs in which all the occurrences of a substring corresponding to more than one pattern are denoted by a single symbol. Thus we would allow 10031 or 20032; however, 10032 and 20031 are disallowed as valid representations of the output. We will therefore require that the output is in a *canonical* form:

Output: A string $Q = q_1 q_2 \ldots q_{n-m+1}$ over the alphabet $\{0, 1, \ldots, v\}$ satisfying the following conditions:

1. $q_i = j > 0$ if P_j matches T at position i. $q_i = 0$ if there is no match.
2. $q_{i_1} = q_{i_2}$ if and only if P_{j_1} matches T at position i_1, P_{j_2} matches T at position i_2, and $P_{j_1} = P_{j_2}$.

For ease of exposition, assume without loss of generality that m divides n, and let $q = n/m$. Furthermore, we assume that all the patterns are distinct. We do this to obtain the output in normal form. There is no loss of generality in making this claim since given a set of patterns $P_1, P_2, \ldots P_v$, we can eliminate duplicates and compute the reduced set consisting of only distinct patterns easily using naming (as in the case of string A before) in linear work and time $O(\log m)$.

Again cut the text T into q nonoverlapping pieces of length m each: $T_1, T_2, \ldots, T_q$. Compute the characteristics of the set consisting of all the prefixes and all the suffixes of the set of strings $\Delta = \{P_1, P_2, \ldots, P_v, T_1, T_2, \ldots, T_q\}$. To derive linear work, the following algorithm is used:

1. Each string in Δ is cut into $m/\log m$ nonoverlapping substrings of length $\log m$. Construct the automaton accepting all these substrings. Each substring is characterized by the name of the state accepting it. Compress the strings, obtaining $(v+q)$ strings of length $m/\log m$ each. Here each string in Δ plays the role of A in the s–p algorithm; this step is analogous to step 1 of the s–p algorithm.
2. For each string in Δ, create $\log m$ strings of length $m/\log m - 1$ each. A symbol in a new string stands for a substring of length $\log m$ in the original string. We obtain $(v+q)\log m$ strings of length $m/\log m - 1$ each. Here each string in Δ plays the role of B in the s–p algorithm; this step is analogous to step 2 of the s–p algorithm.
3. Compute the characteristics of the set consisting of all the proper suffixes of the strings computed in step 1 and all the proper prefixes of the strings computed in step 2. This step is analogous to step 3 of the s–p algorithm.
4. Compute the characteristics of the "remainder" strings. This step is analogous to step 4 of the s–p algorithm.
5. Generalizing from our pattern-matching algorithm, we note that P_i matches T at position $j = \alpha m - \beta$, $1 \le \alpha \le q$, $0 \le \beta \le m-1$, $1 \le j \le n-m+1$, if and only if the suffix of length $(\beta+1)$ of T_α matches the prefix of length $(\beta+1)$ of P_i and the prefix of length $(m-\beta-1)$ of $T_{\alpha+1}$ matches the suffix of length $(m-\beta-1)$ of P_i.

This can be done as sketched below.

Assume the existence of some vector V of length $(2(n + vm) + 1)^2$ initialized to 0. For a string S, let $pref(S, i)$ and $suf(S, i)$ denote the prefix and the suffix of length i of S, respectively.

From the previous steps, we can assume that we have computed the characteristics of all the prefixes and all the suffixes of the patterns and the text pieces. We now proceed in two steps:

(a) We assign a processor to each position of each pattern. In parallel, for each position $\gamma \leq m$ of each pattern P_i, the processor assigned to it writes i in the location $V[\chi(pref(P_i, \gamma)) + (2(n + vm) + 1)\chi(suf(P_i, m - \gamma))]$. As a result of this step, we have "coded" all existing pairs of prefix–suffix for each pattern.

(b) We assign a processor to each position of each T_j. The processor at position $m-\gamma+1$ of T_j reads the value of $V[\chi(suf(T_j, \gamma))+(2(n+vm)+1)\chi(pref(T_{j+1}, m-\gamma))]$. Pattern P_i matches T at position $jm - \gamma + 1$ if and only if the value read was i.

Step 5(a) was needed to make sure that a spurious match is not obtained by combining a prefix of one pattern with the suffix of another pattern.

THEOREM 3.2. *Given a text string of length n and v patterns each of length m, the algorithm above solves the multipattern string-matching problem using $(n+vm)/\log m$ processors in $O(\log m)$ time of a CRCW PRAM.*

3.4. Multitext/multipattern problem.

Input: u *text* strings and v *patterns*. The length of the ith text string is n_i, and the length of all patterns m is the same.

Output: u strings $Q_1, Q_2, \ldots, Q_u$ over the alphabet $\{0, 1, \ldots, v\}$. Each $Q_j = q_{j,1}q_{j,2} \cdots q_{j,n_j-m+1}$ is of length $n_j - m + 1$ and satisfies the following conditions:

1. $q_{k,i} = j > 0$ if and only if P_j matches T_k at position i. (0 indicates no match.)
2. If P_{j_1} matches T_{k_1} at position i_1, P_{j_2} matches T_{k_2} at position i_2, and $P_{j_1} = P_{j_2}$, then $q_{k_1,i_1} = q_{k_2,i_2}$.

THEOREM 3.3. *The obvious modification of the algorithm in* §3.3 *solves the multitext/multipattern matching problem in time $O(\log m)$ using $(vm + \sum_{j=1}^{u} n_j)/\log m$ processors.*

3.5. Multidimensional pattern matching.

We now describe our parallel algorithm for the multidimensional pattern matching problem.

Input: a d-dimensional pattern array $P[1..m, 1..m, \ldots, 1..m]$ of size m^d and a d-dimensional text array $T[1..n, 1..n, \ldots, 1..n]$ of size n^d. ($P[1..m, 1..m, \ldots, 1..m]$ stands for $\{P[i_1, i_2, \ldots, i_d] \mid 1 \leq i_1, i_2, \ldots, i_d \leq m\}$, etc.)

Output: a d-dimensional array $Q[1..n - m + 1, 1..n - m + 1, \ldots, 1..n - m + 1]$ over $\{0, 1\}$ such that $Q[i_1, i_2, \ldots, i_d] = 1$ if and only if the pattern matches the text in position $(i_1, i_2, \ldots, i_d)$, that is, $T[i_1..i_1 + m - 1, i_2..i_2 + m - 1, \ldots, i_d..i_d + m - 1] = P[1..m, 1..m, \ldots, 1..m]$.

Essentially, we use the same framework as given in [Ba78, Bi77, KR87] in the sequential case. The sequential algorithm requires time $O(dn^d)$. In [AL88], a parallel algorithm for this problem was given requiring work $dn^d \log m$ and time $O(d \log m)$. The improvement in complexity in the algorithm implementation we will present comes from the fact that we now overcome the bottleneck in earlier algorithms by using our optimal algorithm for multitext/multipattern string matching from §3.4.

We describe the algorithm recursively:

1. Match the set of strings of the form $P[i_1, i_2, \ldots, i_{d-1}, 1..m]$ in the set of strings of the form $T[k_1, k_2, \ldots, k_{d-1}, j..j+m-1]$. Present the canonical output as several $(d-1)$-dimensional arrays. Specifically, we get the following arrays:

(a) $R[1..m, 1..m, \ldots, 1..m]$. $R[i_1, i_2, \ldots, i_{d-1}]$ is $\chi(P[i_1, i_2, \ldots, i_{d-1}, 1..m])$.

(b) $S_j[1..n, 1..n, \ldots, 1..n]$, $j = 1, 2, \ldots, n-m+1$. $S_j[k_1, k_2, \ldots, k_{d-1}]$ is the characteristic of the string from P matching T at position $(k_1, k_2, \ldots, k_{d-1}, j)$ if such a pattern exists; it is 0 if no such pattern exists.

2. Recursively solve several $(d-1)$-dimensional problems. Specifically, match R in $S_1, S_2, \ldots, S_{n-m+1}$.

P matches T at position $T[k_1, k_2, \ldots, k_{d-1}, j]$ if and only if R matches S_j at position $k_1, k_2, \ldots, k_{d-1}$.

We now proceed to analyze the complexity of the algorithm. From §3.4, we know the following.

FACT 1. *v pattern strings of length m each can be matched in $u \geq v$ text strings of length $n \geq m$ each in work $c \times u \times n$ for an appropriate constant c.*

Using this fact, we need to show the following.

LEMMA 3.4. *For the algorithm above, $W_d(n^d)$, the work required by the text of size n^d, satisfies $W_d(n^d) \leq cdn^d$ and the time required is $O(d \log n)$.*

Proof. Let us review the two steps above. In the first step, we match m^{d-1} pattern strings of length m each in n^{d-1} text strings of length n each. This can be done in work $cn^{d-1}n = cn^d$ and time $O(\log n)$.

In the second step, we solve in parallel $n-m+1$, $(d-1)$-dimensional problems, each consisting of matching a pattern of size m^{d-1} in text of size n^{d-1}. By induction and the previous fact, this can be done in work $(n-m+1)W_{d-1}(n^{d-1}) \leq nc(d-1)n^{d-1} = c(d-1)n^d$. The time required is $O((d-1)\log n)$. Combining the complexities of the two steps, we obtain that the work is $W_d(n^d) \leq cdn^d$ and the time is $O(d \log n)$. □

By interpreting the above lemma appropriately, we have the following result.

THEOREM 3.5. *The algorithm above solves the multidimensional pattern-matching problem in time $O(d \log m)$ using $n^d / \log m$ processors of a CRCW PRAM.*

We now present an example. In this example, $d = 3$, $m = 2$, and $n = 3$. The problem instance is defined by

$$P[1..2, 1..2, 1] = \begin{matrix} a & b \\ b & b \end{matrix},$$

$$P[1..2, 1..2, 2] = \begin{matrix} b & a \\ b & b \end{matrix},$$

$$T[1..3, 1..3, 1] = \begin{matrix} a & b & a \\ b & a & a \\ b & b & b \end{matrix},$$

$$T[1..3, 1..3, 2] = \begin{matrix} b & a & b \\ a & b & b \\ a & b & b \end{matrix},$$

$$T[1..3, 1..3, 3] = \begin{matrix} a & b & a \\ b & b & b \\ a & a & a \end{matrix}$$

(note that P matches T at position $(1,2,2)$).

The recursion has three stages:

1. We compute the characteristics of the appropriate substrings of P. Without loss of generality, they are

$$\begin{aligned}
P[1,1,1..2] &= ab, & \chi(P[1,1,1..2]) &= 1,\\
P[1,2,1..2] &= ba, & \chi(P[1,2,1..2]) &= 2,\\
P[2,1,1..2] &= bb, & \chi(P[2,1,1..2]) &= 3,\\
P[2,2,1..2] &= bb, & \chi(P[2,2,1..2]) &= 3.
\end{aligned}$$

We obtain the following characteristics for the strings of T:

$$\begin{aligned}
T[1,1,1..3] &= aba, & \chi(T[1,1,1..2]) &= 1, & \chi(T[1,1,2..3]) &= 2,\\
T[1,2,1..3] &= bab, & \chi(T[1,2,1..2]) &= 2, & \chi(T[1,2,2..3]) &= 1,\\
T[1,3,1..3] &= aba, & \chi(T[1,3,1..2]) &= 1, & \chi(T[1,3,2..3]) &= 2,\\
T[2,1,1..3] &= bab, & \chi(T[2,1,1..2]) &= 2, & \chi(T[2,1,2..3]) &= 1,\\
T[2,2,1..3] &= abb, & \chi(T[2,2,1..2]) &= 1, & \chi(T[2,2,2..3]) &= 3,\\
T[2,3,1..3] &= abb, & \chi(T[2,3,1..2]) &= 1, & \chi(T[2,3,2..3]) &= 3,\\
T[3,1,1..3] &= baa, & \chi(T[3,1,1..2]) &= 2, & \chi(T[3,1,2..3]) &= 0,\\
T[3,2,1..3] &= bba, & \chi(T[3,2,1..2]) &= 3, & \chi(T[3,2,2..3]) &= 2,\\
T[3,3,1..3] &= bba, & \chi(T[3,3,1..2]) &= 3, & \chi(T[3,3,2..3]) &= 2.
\end{aligned}$$

The pattern is now coded as a two-dimensional object $R[1..2,1..2]$; $R[i,j]$ stands for the characteristic of $P[i,j,1..2]$. The text is coded as two two-dimensional objects $S_1[1..3,1..3]$ and $S_2[1..3,1..3]$; $S_k[i,j]$ stands for the characteristic of $T[i,j,k..k{+}m{-}1]$.

$$R[1..2,1..2] = \begin{matrix} 1 & 2 \\ 3 & 3 \end{matrix},$$

$$S_1[1..3,1..3] = \begin{matrix} 1 & 2 & 1 \\ 2 & 1 & 1 \\ 2 & 3 & 3 \end{matrix},$$

$$S_2[1..3,1..3] = \begin{matrix} 2 & 1 & 2 \\ 1 & 3 & 3 \\ 0 & 2 & 2 \end{matrix}$$

(note that R matches S_2 at position $(1,2)$).

2. As stated in the description of the algorithm, we should now solve two matching problems: R in S_1 and R in S_2. It is simpler to combine these two problems into the single problem of matching R in S_1 and S_2.

We compute the characteristics of the appropriate substrings of R. Without loss of generality, they are

$$\begin{aligned}
R[1,1..2] &= 12, & \chi(R[1,1..2]) &= 1,\\
R[2,1..2] &= 33, & \chi(R[2,1..2]) &= 2.
\end{aligned}$$

We obtain the following characteristics for the strings of S_1 and S_2:

$$\begin{aligned}
S_1[1,1..3] &= 121, & \chi(S_1[1,1..2]) &= 1, & \chi(S_1[1,2..3]) &= 0,\\
S_1[2,1..3] &= 211, & \chi(S_1[2,1..2]) &= 0, & \chi(S_1[2,2..3]) &= 0,\\
S_1[3,1..3] &= 233, & \chi(S_1[3,1..2]) &= 0, & \chi(S_1[3,2..3]) &= 2,\\
S_2[1,1..3] &= 212, & \chi(S_2[1,1..2]) &= 0, & \chi(S_2[1,2..3]) &= 1,\\
S_2[2,1..3] &= 133, & \chi(S_2[2,1..2]) &= 0, & \chi(S_2[2,2..3]) &= 2,\\
S_2[3,1..3] &= 022, & \chi(S_2[3,1..2]) &= 0, & \chi(S_2[3,2..3]) &= 0.
\end{aligned}$$

The pattern is now coded as a one-dimensional object $U[1..2]$; $U[i]$ stands for the characteristic of $R[i,1..2]$. The text is coded as four one-dimensional objects $V_{1,1}[1..3]$, $V_{1,2}[1..3], V_{2,1}[1..3], V_{2,2}[1..3]$; $V_{j,k}[i]$ stands for the characteristic of $S_j[i,k..k+1]$. (U and V play the same role as R and S in the previous stage.)

$$U[1..2] = 12,$$

$$V_{1,1}[1..3] = 100,$$

$$V_{1,2}[1..3] = 002,$$

$$V_{2,1}[1..3] = 000,$$

$$V_{2,2}[1..3] = 120$$

(note that U matches $V_{2,2}$ at position 1).

3. We should now solve four matching problems. Again we combine them into a single problem.

We compute the characteristics of the appropriate substrings of U. Without loss of generality, they are

$$U[1..2] = 12, \quad \chi(U[1..2]) = 1.$$

We obtain the following characteristics for the strings of $V_{1,1}, V_{1,2}, V_{2,1}, V_{2,2}$:

$$\begin{array}{lll} V_{1,1}[1..3] = 100, & \chi(V_{1,1}[1..2]) = 0, & \chi(V_{1,1}[2..3]) = 0, \\ V_{1,2}[1..3] = 002, & \chi(V_{1,2}[1..2]) = 0, & \chi(V_{1,2}[2..3]) = 0, \\ V_{2,1}[1..3] = 000, & \chi(V_{2,1}[1..2]) = 0, & \chi(V_{2,1}[2..3]) = 0, \\ V_{2,2}[1..3] = 120, & \chi(V_{2,2}[1..2]) = 1, & \chi(V_{2,2}[2..3]) = 0. \end{array}$$

In stage 3, we found that U matches $V_{2,2}$ at position 1. Therefore, the output of stage 2 is that R matches S_2 at position (1,2). Hence in stage 1, we indeed deduce that P matches T at position $(1,2,2)$. Of course, the answer could be written in the form of Q as required by the formal specifications of the output.

3.6. The pattern-occurrence detection. In this section, we consider the problem of string matching when the text has been cut into a number of pieces. Formally, we have the following:

Input: a *pattern* string $P = p_0p_1 \dots p_{m-1}$ of length m and l *distinct text substrings* $T_1, T_2, \dots, T_l$ of length k each.

Output: decide whether there exists a permutation of the text substrings for which the pattern is a match and, if yes, produce one such permutation and the match position for it.

Before we proceed with the description of our algorithm, several remarks are in order. Generally, one might consider that the strings are of arbitrary lengths and possibly replicated. Then the "assembling" of strings into a single text string is difficult. The deterministic problem of pattern matching in this case is NP-hard [TU88]. Approximation algorithms for this more difficult problem, which appeared in molecular biology [TU88], were given in [CD88, TU88, KM95, Tur89, Ukk90, BJLTY91].

We now proceed with the description of our algorithm solving the simpler problem we have formally defined above. To simplify the exposition, consider the case when $m \geq 3k$ and m is a multiplication of k. Observe that a matching permutation exists if and only if there exist *distinct* $j_1, j_2, \dots, j_s$, where $j_s \geq 3$ such that the pattern is equal to a suffix of T_{j_1} followed by $T_{j_2}, \dots, T_{j_{s-1}}$ followed by a prefix of T_{j_s}.

The algorithm proceeds in two steps.

1. We compute an integer vector $CENTER[0, \dots, m-1]$. $CENTER[i] = j$ if and only if $p_i p_{i+1} \dots p_{i+k-1} = t_{j,0} t_{j,1} \dots t_{j,k-1}$; if no such j exists, set $CENTER[i] = 0$.

2. We compute a bit vector $ENDS[0..k-1]$. $ENDS[0] = 1$. For $i \geq 1$, $ENDS[i] = 1$ if and only if there exist distinct j_a and j_b such that
(a) $suf(T_{j_a}, i) = pref(P, i)$ ($p_0 \dots p_{i-1}$ is equal to the suffix of length i of T_{j_a}.),
(b) $pref(T_{j_b}, k-i) = suf(P, k-i)$,
(c) for each $r = 0, 1, \dots, m/k - 2$, $p_{i+rk} \dots p_{i+(r+1)k-1}$ is different from both T_{j_a} and T_{j_b}.

A match exists if and only if for some $i = 0, 1, \dots, k-1$, (i) $ENDS[i] = 1$ and (ii) all $CENTER[i + rk]$ for $r = 0, 1, \dots, m/k - 2$ are $\neq 0$ and distinct.

All these vectors can be computed in time $O(\log m)$ using $O(kl)$ work. It is easy to see how to complete the algorithm to produce the output in those complexity bounds too.

If $m < k$, we also have to test whether the pattern matches one of the text pieces, which too can be done in optimal speedup. Therefore, we have the following result.

THEOREM 3.6. *The above algorithm solves the pattern-occurrence-detection problem in time $O(\log m)$ using $kl/\log m$ processors on a CRCW PRAM.*

3.7. On-line string matching. In this section, we consider the parallel version of the on-line string-matching problem.

On-line Input: a pattern string P and a sequence of l *text substrings* $T_1, T_2, \dots, T_l$ given dynamically.

On-line Output: for each i, $i = 1, 2, \dots, l$, after producing the output for $T_1, T_2, \dots, T_{i-1}$, produce the output consisting of all those positions in which P matched $T_1 T_2 \dots T_{i-1} T_i$ and that were not reported previously.

To specify the complexity of an on-line parallel algorithm, we will need to introduce a few notions. Let m denote the length of P and for each i, let k_i denote the length of T_i and let $n_i = \sum_{j=1}^{i} k_j$. Now let i be in $\{1, 2, \dots, l\}$. Consider the time instant when the algorithm finished processing the text substrings $T_1, T_2, \dots, T_{i-1}$ and produced the corresponding output. It is now given T_i and produces the "incremental" output. Let $T(n_{i-1}, m, k_i)$ and $P(n_{i-1}, m, k_i)$, respectively, denote the time and the number of processors required by the algorithm to produce that output. The total *amortized work* done by the algorithm is defined as $\sum_{i=1}^{l} P(n_{i-1}, m, k_i) T(n_{i-1}, m, k_i)$. We say that this algorithm has *optimal amortized speedup* provided that for all possible inputs the amortized work done is within a constant factor away from the time complexity of solving this problem by the best-known sequential algorithm.

We will now sketch an optimal-speedup (on-line) algorithm briefly. The details are easy to fill out. To simplify the exposition, consider only the case when pattern matching was done for some pattern and some string, and then the string is further extended and pattern matching needs to be done for the new longer string. Assume then that, using our algorithm from §2, pattern matching has been solved for a pattern of length m and a text string S of some length n, and then an additional text string

S' of length k is presented. The main idea behind the on-line implementation is to use the algorithm and data structures used in the off-line case described in §2, but handling each extra text chunk as it is given to the to the algorithm dynamically. Of course, it is important to be able to do this *without* recomputing most of the information that was done earlier on. The work bounds are estimated by amortizing on text chunks that are multiples of $\log m$.

We must now solve the pattern matching problem for the augmented string SS'. In general, assuming that $n > 2m$, we can write $S = S_1S_2S_3$, where $length(S_1)$ is a positive multiple of m, $length(S_2)$ equals m, and $length(S_3)$ is smaller than m. We have all the suffix information for S_2 and all the prefix information for S_3.

We consider two cases:

1. $length(S_3) + k < m$. To avoid simple case analysis, assume $k \geq \log m$. We need to extend the prefix information available for S_3 to get the prefix information for S_3S'. By applying the automaton M, it is possible to do so in work of $O(k)$ in time of $O(\log m)$. (By refining our algorithm, the total time could be lower for certain cases where k is smaller than m, but it is not worth considering this here.)

2. $length(S_3) + k \geq m$. Write $S' = S_aS_b$, where $length(S_a) = m - length(S_3)$. We will need to compute the following:

(a) The prefix information for S_3S_a. This is done as described above.

(b) The suffix information for S_3S_a. This is done as in our algorithm for regular string matching, in time $O(\log m)$ and work $O(m)$.

(c) The prefix information for S_b. This is also done as described above.

THEOREM 3.7. *The algorithm above solved the on-line pattern-occurrence problem in parallel amortized linear work. The time for processing each substring is* $O(\log m)$.

4. Conclusions. In this paper, we employ s–p matching as the core computation in several pattern- and string-matching problems. Our main result is a parallel algorithm for computing s–p matching which has optimal speedup on a CRCW PRAM. This algorithm is based on novel techniques that combine notions of characteristic functions with the well-known automaton-based approach to string matching that uses failure functions. Briefly, we first break the text and pattern (both of length m) appropriately into "small pieces" of size $O(\log m)$. Then, using a parallel variant of the algorithm due to Aho and Corasick [AC75] for short ($\log m$-length) strings, we group these small pieces into equivalence classes based on string equality. Given such equivalence classes, we assemble these small pieces together and solve the problem on the entire input. This is done by successively and consistently refining the equivalence classes. Using this algorithm for s–p matching as the basic building block, we specify optimal-speedup parallel algorithms for several pattern- and string-matching problems.

REFERENCES

[ABF93] A. AMIR, G. BENSON, AND M. FARACH, *Optimal parallel two dimensional pattern matching*, in Proc. 5th ACM Symposium on Parallel Algorithms and Architectures, Association for Computing Machinery, New York, 1993, pp. 79–85.

[AC75] A. V. AHO AND M. J. CORASICK, *Efficient string matching*, Comm. Assoc. Comput. Mach., 18 (1975), pp. 333–340.

[AILSV88] A. APOSTOLICO, C. ILIOPOULOS, G. M. LANDAU, B. SCHIEBER, AND U. VISHKIN, *Parallel construction of a suffix tree with applications*, Algorithmica, 3 (1988), pp. 347–365.

[AL88] A. AMIR AND G. M. LANDAU, *Fast parallel and serial multi dimensional approximate array matching*, Theoret. Comput. Sci., 81 (1991), pp. 97–115.

[Ba78] T. P. Baker, *A technique for extending rapid exact-match string matching to arrays of more than one dimension*, SIAM J. Comput., 7 (1978), pp. 533–541.

[Bi77] R. S. Bird, *Two dimensional pattern matching*, Inform. Process. Lett., 6 (1977), pp. 168–170.

[Br74] R. P. Brent, *The parallel evaluation of general arithmetic expressions*, J. Assoc. Comput. Mach., 21 (1974), pp. 201–206.

[BG90] D. Breslauer and Z. Galil, *An optimal $O(\log\log n)$ time parallel string matching algorithm*, SIAM J. Comput., 19 (1990), pp. 1051–1058.

[BJLTY91] A. Blum, T. Jiang, M. Li, J. Tromp, and M. Yanakakis, *Linear approximation of shortest superstrings*, in Proc. 23rd ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1991, pp. 328–336.

[BM77] R. S. Boyer and J. S. Moore, *A fast string searching algorithm*, Comm. Assoc. Comput. Mach., 20 (1977), pp. 762–772.

[CCG+93] R. Cole, M. Crochemore, Z. Galil, L. Gasieniec, R. Hariharan, S. Muthukrishnan, K. Park, and W. Rytter, *Optimally fast parallel algorithms for preprocessing and pattern matching in one and two dimensions*, in Proc. 34th Annual IEEE Conference on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1993, pp. 248–258.

[CD88] J. L. Cornette and C. Delisi, *Some mathematical aspects of mapping DNA cosmids*, Cell Biophysics, 12 (1988), pp. 271–293.

[FL80] M. J. Fischer and L. Ladner, *Parallel prefix computation*, J. Assoc. Comput. Mach., 27 (1980), pp. 831–838.

[G84] Z. Galil, *Optimal parallel algorithms for string matching*, Inform. and Control, 67 (1985), pp. 144–157.

[G92] ———, *A constant-time optimal parallel String-Matching Algorithm*, in Proc. 23rd ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1992, pp. 69–76.

[GS83] Z. Galil and J. I. Seiferas, *Time-space optimal string matching*, J. Comput. System Sci., 26 (1983), pp. 280–294 and 338–355.

[H88] T. Hagerup, *On saving space in parallel computation*, Inform. Process. Lett., 29 (1988), pp. 327–329.

[J92] J. Já Já, *An Introduction to Parallel Algorithms,* Addison–Wesley, Reading, MA, 1992.

[KM95] J. D. Kececioglu and E. W. Myers, *Combinatorial algorithms for DNA sequence assembly*, Algorithmica, 13 (1995), pp. 7–51.

[KMP77] D. E. Knuth, J. H. Morris, and V. R. Pratt, *Fast pattern matching in strings*, SIAM J. Comput., 6 (1977), pp. 323–350.

[KMR72] R. M. Karp, R. E. Miller, and A. L. Rosenberg, *Rapid identification of repeated patterns in strings, trees, and arrays*, in Proc. 4th ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, (1972), pp. 125–136.

[KP92] Z. M. Kedem, and K. V. Palem, *Optimal parallel algorithms for forest and term matching*, Theoret. Comput. Sci., 93 (1992), pp. 245–264.

[KR87] R. M. Karp and M. O. Rabin, *Efficient randomized pattern-matching algorithms*, IBM J. Res. Develop., 31 (1987), pp. 249–260.

[M88] T. R. Mathies, *A fast parallel algorithm to determine edit distance*, Technical report CMU-CS-88-130, Department of Computer Science, Carnegie Mellon University, Pittsburgh, 1988.

[TU88] J. Tarhio and E. Ukkonen, *A greedy approximation algorithm for constructing shortest common superstrings*, Theoret. Comput. Sci., 57 (1988), pp. 131–145.

[Tur89] J. Turner, *Approximation algorithms for the shortest common superstring problem*, Inform. and Comput., 83 (1989) pp. 1–20.

[Ukk90] E. Ukkonen, *A linear time algorithm for finding approximate shortest common superstrings*, Algorithmica, 5 (1990), pp. 313–323.

[V85] U. Vishkin, *Optimal parallel pattern matching in strings*, Inform. and Control, 67 (1985), pp. 91–113.

[V91] ———, *Deterministic sampling: A new technique for fast pattern matching*, SIAM J. Comput., 20 (1992), pp. 22–40.

[W73] P. Weiner, *Linear pattern matching algorithm*, in Proc. 14th IEEE Symposium on Switching and Automata Theory, IEEE Computer Society Press, Los Alamitos, CA, 1973, pp. 1–11.

SIAM J. COMPUT.
Vol. 25, No. 5, pp. 1024–1044, October 1996

RANDOMIZED CONSENSUS IN EXPECTED $O(N\log^2 N)$ OPERATIONS PER PROCESSOR*

JAMES ASPNES† AND ORLI WAARTS‡

Abstract. This paper presents a new randomized algorithm for achieving consensus among asynchronous processors that communicate by reading and writing shared registers. The fastest previously known algorithm requires a processor to perform an expected $O(n^2 \log n)$ read and write operations in the worst case. In our algorithm, each processor executes at most an expected $O(n \log^2 n)$ read and write operations, which is close to the trivial lower bound of $\Omega(n)$.

All previously known polynomial-time consensus algorithms were structured around a *shared-coin* protocol [*J. Algorithms*, 11 (1990), pp. 441–446] in which each processor repeatedly adds random ±1 votes to a common pool. Consequently, in all of these protocols, the worst-case expected bound on the number of read and write operations done by a single processor is asymptotically no better than the bound on the total number of read and write operations done by all of the processors together. We succeed in breaking this tradition by allowing the processors to cast votes of increasing weights. This grants the adversary greater control since he can choose from up to n different weights (one for each processor) when determining the weight of the next vote to be cast. We prove that our shared-coin protocol is nevertheless correct using martingale arguments.

Key words. consensus, distributed algorithms, shared memory, randomized algorithms, asynchronous computation, martingales

AMS subject classifications. Primary 68Q22; Secondary 60G42, 60F10

1. Introduction.

In the *consensus* problem, each of n asynchronous processors starts with an input value 0 or 1 not known to the others and runs until it chooses a *decision value* and halts. The protocol must be *consistent*: no two processors choose different decision values; *valid*: the decision value is some processor's input value; and *wait-free*: each processor decides after a finite expected number of its *own* steps regardless of other processors' halting failures or relative speeds.

We consider the consensus problem in the standard model of asynchronous shared-memory systems. The processors communicate via a set of single-writer, multireader atomic registers. Each such register can be written by only one processor, its owner, but all processors can read it. Reads and writes to such a register can be viewed as occurring at a single instant of time.

Consensus is fundamental to synchronization without mutual exclusion and hence lies at the heart of the more general problem of constructing highly concurrent data structures [20]. It can be used to obtain wait-free implementations of arbitrary abstract data types with atomic operations [20, 23]. Consensus is also *complete* for *distributed decision tasks* [11] in the sense that it can be used to solve all such tasks that have a wait-free solution.

Consensus is often viewed as a game played between a set of processors and an adversary scheduler. Using the standard wait-free model of an asynchronous shared-

* Received by the editors November 30, 1992; accepted for publication (in revised form) January 25, 1994. A preliminary version of this work appeared in *Proc.* 33rd *IEEE Symposium on Foundations of Computer Science.*

† Department of Computer Science, Yale University, New Haven, CT 06520-8285 (aspnes @cs.yale.edu). During the time of this research, the first author was at Carnegie–Mellon University and supported in part by an IBM Graduate Fellowship.

‡ Computer Science Division, University of California at Berkeley, Berkeley, CA 94720 (waarts@cs.berkeley.edu). During the time of this research, the second author was at Stanford University and supported in part by an IBM Graduate Fellowship, U.S. Army Research Office grant DAAL-03-91-G-0102, and NSF grant CCR-8814921.

memory system, each processor can execute as an atomic step either (a) a single read or write operation or (b) a flip of a local fair coin not visible to the other processors. The sequencing of the processors' actions is controlled by a *scheduler*, defined as a function that at each step selects a processor to run based on the entire prior history of the system, including the internal states of the processors. (Concurrency is modeled by interleaving.) Remarkably, it has been shown that the ability of the scheduler to stop even a single processor is sufficient to prevent consensus from being solved by a deterministic algorithm [10, 12, 16, 20, 22]. Nevertheless, it can be solved by *randomized* protocols in which each processor is guaranteed to decide after a finite *expected* number of steps.

Chor, Israeli, and Li [10] provided the first solution to the problem, but their solution deviated from the standard model by assuming that the processor can flip a coin and write the result in a single atomic step. Abrahamson [1] demonstrated that consensus is possible even for the standard model, but his protocol required an exponential expected number of steps. Since then, a number of polynomial-work consensus protocols have been proposed. Protocols that use unbounded registers have been proposed by Aspnes and Herlihy [4] (the first polynomial-time algorithm), Saks, Shavit, and Woll [24] (optimized for the case where processors run in lock step), and Bracha and Rachman [8] (running time $O(n^2 \log n)$). Protocols that use bounded registers have been proposed Attiya, Dolev, and Shavit [5] (running time $O(n^3)$), Aspnes [3] (running time $O(n^2(p^2+n))$, where p is the number of active processors), Bracha and Rachman [7] (running time $O(n(p^2+n))$), and Dwork, Herlihy, Plotkin, and Waarts [13] (immediate application of what they call time-lapse snapshots and with the same running time as [7]).

The main goal of a wait-free algorithm is usually to minimize the worst-case expected bound on the work done by a single processor. Still, for all of the known polynomial-work wait-free consensus protocols, the worst-case expected bound on the work done by a single processor is asymptotically no better than the bound on the total work done by all of the processors together.

Therefore, one of the main contributions of this paper is in showing that wait-free consensus can be solved without requiring the fast processors to perform much more than their fair share of the worst-case total amount of work executed by all processors together. At the same time, we improve significantly on the complexity of all currently known wait-free consensus protocols, obtaining a protocol in which a processor executes at most an expected $O(n \log^2 n)$ read and write operations, which is close to the trivial lower bound of $\Omega(n)$.[1] To do this, we introduce a new *weak shared-coin* protocol [4] that is based on a combination of the shared-coin protocol described by Bracha and Rachman [8] and a new technique called *weighted voting*, where votes of faster processors carry more weight.[2] We believe that our weighted-voting technique will find applications in other wait-free shared-memory problems such as approximated consensus and resource allocation.

The rest of the paper is organized as follows. Section 2 describes the intuition behind our solution while emphasizing the main difference between our solution and those in [3, 4, 5, 7, 8, 13, 24]. Section 3 describes our shared-coin protocol. Section 4 reviews martingales and derives some of their properties. Section 5 contains the proof

[1] As discussed in §6, this gain in per-processor performance involves a slight increase in the total work performed by all processors when compared with the Bracha–Rachman protocol.

[2] The consensus protocol can be constructed around our shared-coin protocol using the established techniques of Aspnes and Herlihy [4].

of correctness of our shared-coin protocol. A discussion of the results appears in §6.

2. Intuition and relation to previous results.

All of the known polynomial-work consensus protocols are based on the same primitive, the *weak shared coin.* A weak shared coin returns a single bit to each processor; for each possible value $b \in \{0, 1\}$, the probability that all processors see b must be at least a constant δ (the *agreement parameter* of the coin), regardless of scheduler behavior.[3] Aspnes and Herlihy [4] showed that given a weak shared coin with constant agreement parameter, it is possible to construct a consensus protocol by executing the coin repeatedly within a rounds-based framework which detects agreement. The number of operations executed by each processor in this construction is $O\left(\left(n + T\left(n\right)\right)/\delta\right)$, where $T(n)$ is the expected work per processor for the weak-shared-coin protocol. For constant δ and under the reasonable assumption that $T(n)$ dominates n, the work per processor to achieve consensus becomes simply $O\left(T\left(n\right)\right)$.

Therefore, to construct a fast consensus protocol, one need only construct a fast weak shared coin. The underlying technique for building a weak shared coin has not changed substantially since the protocol described in [4]; each processor repeatedly adds random ± 1 votes to a common pool until either the total vote is far from the origin [3, 4, 5, 7, 13] or a predetermined number of votes have been cast [8, 24]. Any processor that sees a nonnegative total vote decides 1, and those that see a negative total vote decide 0. (The differences between the protocols are largely in how termination is detected and how the counter for the vote is implemented.)

There are many advantages to this approach. The processors effectively act as anonymous conduits of a stream of unpredictable random increments. If the scheduler stops a particular processor, at worst all it does is keep one vote from being written out to the common pool—the next local coin flip executed by some other processor is no more or less likely to give the value the scheduler wants than the next one executed by the processor it has just stopped. Intuitively, the scheduler's power over the outcome of the shared coin is limited to filtering out up to $n - 1$ local coin flips from this stream of independent random variables. But the effect of this filtering is at worst equivalent to adjusting the final tally of votes by up to $n - 1$. If a constant multiple of n^2 votes are cast, the total variance will be $\Omega(n^2)$, and using a normal approximation the protocol can guarantee that with constant probability the total vote is more than n away from the origin, rendering the scheduler's adjustment ineffective.

Alas, the very anonymity of the processors that is the strength of the voting technique is also its greatest weakness. To overcome the scheduler's power to withhold votes, it is necessary that a total of $\Omega(n^2)$ votes are cast— but the scheduler might also choose to stop all but one of the processors, leaving that lone processor to generate all $\Omega(n^2)$ votes by itself. Consequently, for all of the polynomial-work wait-free consensus protocols currently known, the worst-case expected bound on the work done by a single processor is asymptotically no better than the bound on the total work done by all of the processors together.

We overcome this problem by modifying the $O(n^2 \log n)$ protocol of Bracha and Rachman [8] to allow the processor to cast votes of increasing weight. Thus a fast processor or a processor running in isolation can quickly generate votes of sufficient total variance to finish the protocol, at the cost of giving the scheduler greater control

[3] The term *agreement parameter* was first used by Saks et al. [24] in place of the more melodramatic but less descriptive term *defiance probability* of Aspnes and Herlihy [4]. Aspnes [3] used a *bias* parameter, equal to 1/2 minus the agreement parameter; however, this quantity is not as useful as the agreement parameter in the context of a multiround consensus protocol.

```
PROCEDURE shared_coin()
begin
    $my\_reg(variance, vote) \leftarrow (0, 0)$
    $t \leftarrow 1$
    repeat
        for $i = 1$ to $c$ do
            $vote \leftarrow local\_flip() \times w(t)$
            $my\_reg \leftarrow (my\_reg.variance + w(t)^2, my\_reg.vote + vote)$
            $t \leftarrow t + 1$
        end
        read all the registers, summing the variance fields into the local
        variable total_variance
    until $total\_variance > K$
    read all the registers, summing the vote fields into the local variable
    total_vote
    if $total\_vote > 0$
    then output 1
    else output 0
end
```

FIG. 1. *Shared-coin protocol.*

by allowing it both to withhold votes with larger impact and to choose among up to n different weights (one for each processor) when determining the weight of the next vote to be cast.

There are two main difficulties that this approach entails; the first is that careful adjustment of the weight function and other parameters of the protocol is necessary to make sure that it performs correctly. More importantly, correctness proofs for previous shared coins based on random walks or voting [3, 4, 5, 7, 8, 13, 24] considered only equally weighted votes and have therefore been able to treat the sequence of votes as a sequence of independent random variables using a substitution argument. Because our protocol allows the weight of the ith vote to depend on which processor the scheduler chooses to run, which may depend on the outcomes of previous votes, we cannot assume independence.

However, the *sign* of each vote is determined by a fair coin flip that the scheduler cannot predict in advance, and so despite all the scheduler's powers, the expected value of each vote before it is cast is always 0. This is the primary requirement of a *martingale process* [6, 15, 21]. Under the right conditions, martingales have many similarities to sequences of sums of independent random variables. In particular, martingale analogues of the central limit theorem and Chernoff bounds will be used in the proof of correctness.

3. The shared-coin protocol.

Figure 1 gives pseudocode for each processor's behavior during the shared-coin protocol. Each processor repeatedly flips a local coin that returns the values $+1$ and -1 with equal probability. The weighted value of each flip is $w(t)$ or $-w(t)$, respectively, where t is the number of coins flipped by the processor up to and including its current flip. Each weighted flip represents a vote for either the output value 1 (if positive) or 0 (if nonpositive). After each flip, the processor updates its register to hold the sum of the weighted flips it has performed and the sum of the squares of their values. After every c flips, the processor reads the registers of all the other processors and computes the sum of all the weighted flips (the total vote) and the sum of the squares of their values (the total variance). If the total variance is greater than the quorum K, it stops and outputs 1 if the total vote is positive and 0 otherwise. Alternatively, if the total variance has not yet reached the quorum K, it continues to flip its local coin.

The function *local_flip* returns the values 1 and -1 randomly with equal probability. The values K and c are parameters of the protocol which will be set depending on the number of processors n to give the desired bounds on the agreement parameter and running time. The weight function $w(t)$ is used to make later local coin flips have more effect than earlier ones so that a processor running in isolation will be able to achieve the quorum K quickly. The weight function will be assumed to be of the form $w(t) = t^a$, where a is a nonnegative parameter depending on n; although other weight functions are possible, this choice simplifies the analysis.

We will demonstrate that for a suitable choice of K, c, and a, all processors return 1 with constant probability; the case of all processors returning 0 will follow by symmetry. The structure of the argument follows the proof of correctness of the less sophisticated protocol of Bracha and Rachman [8], which corresponds to Figure 1 when $w(t)$ is the constant 1, $K = \Theta(n^2)$, and $c = \Theta(n/\log n)$. Votes cast before the quorum K is reached will form a pool of *common votes* that all processors see.[4] We will show that with constant probability, (i) the total of the common votes is far from the origin and (ii) the sum of the *extra votes* cast between the time the quorum is reached and the time some processor does its final read in line 13 is small so that the total vote read by each processor will have the same sign as the total common vote.

This simple overview of the proof hides many tricky details. To simplify the analysis, we will concentrate not on the votes actually written to the registers but on the votes whose values have been *decided* by the processors' execution of the local coin flip in line 7; conversion back to the values actually in the registers will be done by showing a bound on the difference between the total decided vote and the total of the register values. In effect, we are treating a vote as having been "cast" the moment that its value is determined instead of when it becomes visible to the other processors.

Some care is also needed to correctly model the sequence of votes. Most importantly, as pointed out above, allowing the weight of the ith vote to depend on which processor the scheduler chooses to run means the votes are not independent. Thus the straightforward proof techniques used for protocols based on a stream of identically distributed random votes no longer apply, and it is necessary to bring in the theory of martingales to describe the execution of the protocol.

4. Martingales.

A *martingale* is a sequence of random variables $S_1, S_2, \ldots$ which informally may be thought of as representing the changes in the fortune of a gambler playing in a fair casino. Because the gambler can choose how much to bet or which game to play at each instant, each random variable S_i may depend on all previous events. But because the casino is fair and the gambler cannot predict the future, the expected change in the gambler's fortune at any play is always 0.

We will need to use a very general definition of a martingale [6, 15, 21]. The simplest definition of a martingale says that the expected value of S_{i+1} given $S_1, S_2, \ldots, S_i$

[4] The definitions of the common and extra votes we will use differ slightly from those used in [8]; the formal definitions appear in §5.

is just S_i. To use a gambling analogy, this definition says that a gambler who knows only the previous values of her fortune cannot predict its expected future value any better than by simply using its current value. But what if the gambler knows more information than just the changing size of her bankroll? For example, imagine that she is placing bets on a fair version of roulette and always bets on either red or black. Knowing that her fortune increased after betting red will tell her only that one of eighteen red numbers came up; but a real gambler will see precisely *which* of the eighteen numbers it was. Still, we would like to claim that this additional knowledge does not affect her ability to predict the future. To do so, the definition of a martingale must be extended to allow additional information to be represented explicitly.

The tool used to represent the information known at any point in time will be a concept from measure theory, a *σ-algebra.*[5] The description given here is informal; more complete definitions can be found in [15, §§IV.3, IV.4, and V.11] or [6].

4.1. Knowledge, σ-algebras, and measurability. Recall that any probabilistic statement is always made in the context of some (possibly implicit) *sample space.* The elements of the sample space (called *sample points*) represent all possible results of some set of experiments, such as flipping a sequence of coins or choosing a point at random from the unit interval. Intuitively, all randomness is reduced to selecting a single point from the sample space. An *event*, such as a particular coin flip coming up heads or a random variable taking on the value 0, is simply a subset of the sample space that "occurs" if one of the sample points it contains is selected.

If we are omniscient, we can see which sample point is chosen and thus can tell for each event whether it occurs or not. However, if we have only partial information, we will not be able to determine whether some events occurred or not. We can represent the extent of our knowledge by making a list of all events we do know about. This list will have to satisfy certain closure properties; for example, if we know whether or not A occurred and whether or not B occurred then we should know whether or not the event "A or B" occurred.

We will require that the set of known events be a *σ-algebra.* A σ-algebra $\mathcal{F}$ is a family of subsets of a sample space Ω that (i) contains the empty set; (ii) is closed under complement: if $\mathcal{F}$ contains A, it contains $\Omega \setminus A$ (the *complement* of A); and (iii) is closed under countable union: if $\mathcal{F}$ contains all of $A_1, A_2, \ldots$, it contains $\bigcup_{i=1}^{\infty} A_i$.[6] An event A is said to be *$\mathcal{F}$-measurable* if it is contained in $\mathcal{F}$. In our context, the term "measurable," which comes from the original measure-theoretic use of σ-algebras to represent families of sets on which a probability distribution is well defined, simply means "known."

We "know" about an event if we can determine whether or not it occurred. What about random variables? A random variable X is defined to be *$\mathcal{F}$-measurable* if every event of the form $X \leq c$ is $\mathcal{F}$-measurable. (The closure properties of $\mathcal{F}$ then imply that such events as $a \leq X < b$, $X = d$, and so forth are also $\mathcal{F}$-measurable.) Looking at the situation in reverse, given random variables $X_1, X_2, \ldots$, we can consider the minimum σ-algebra $\mathcal{F}$ for which each of the random variables is $\mathcal{F}$-measurable; this σ-algebra, written $\langle X_i \rangle$, is called the σ-algebra *generated* by $X_1, X_2, \ldots$ and represents all information that can be inferred from knowing the values of the generators.

A σ-algebra gives us a rigorous way to define "knowledge" in a probabilistic context. Measurability and generated σ-algebras give us a way to move back and

[5] This is sometimes called a *σ-field.*

[6] Additional properties, such as being closed under finite union or intersection, follow immediately from this definition.

forth between the abstract concept of a σ-algebra and concrete statements about which random variables are completely known. To analyze random variables that are only *partially* known, we need one more definition. We need to extend conditional expectations so that the condition can be a σ-algebra rather than just a collection of random variables.

For each event A, let I_A be the indicator variable that is 1 if A occurs and 0 otherwise. Let $U = \mathrm{E}[X \mid \mathcal{F}]$ be a random variable such that (i) U is $\mathcal{F}$-measurable and (ii) $\mathrm{E}[UI_A] = \mathrm{E}[XI_A]$ for all A in $\mathcal{F}$. The random variable $\mathrm{E}[X \mid \mathcal{F}]$ is called the *conditional expectation* of X with respect to $\mathcal{F}$ [15, §V.11]. Intuitively, the first condition on $\mathrm{E}[X \mid \mathcal{F}]$ says that it reveals no information not already found in $\mathcal{F}$. The second condition says that just knowing that some event in $\mathcal{F}$ occurred does not allow one to distinguish between X and $\mathrm{E}[X \mid \mathcal{F}]$; this fact ultimately implies that $\mathrm{E}[X \mid \mathcal{F}]$ uses all information that is found in $\mathcal{F}$ and is relevant to X.

If $\mathcal{F}$ is generated by random variables $X_1, X_2, \ldots$, the conditional expectation $\mathrm{E}[X \mid \mathcal{F}]$ reduces to the simpler version $\mathrm{E}[X \mid X_1, X_2, \ldots]$. Some other facts about conditional expectation that we will use (but not prove) are the following: if X is $\mathcal{F}$-measurable, then $\mathrm{E}[XY \mid \mathcal{F}] = X\,\mathrm{E}[Y \mid \mathcal{F}]$ (which implies $\mathrm{E}[X \mid \mathcal{F}] = X$); and if $\mathcal{F}' \subseteq \mathcal{F}$, then $\mathrm{E}[\mathrm{E}[X \mid \mathcal{F}] \mid \mathcal{F}'] = \mathrm{E}[X \mid \mathcal{F}']$. See [15, §V.11].

4.2. Definition of a martingale. We now have the tools to define a martingale when the information available at each point in time is not limited to just the values of earlier random variables. A *martingale* $\{S_i, \mathcal{F}_i\}, 1 \le i \le n$, is a stochastic process where each S_i is a random variable representing the state of the process at time i and $\mathcal{F}_i$ is a σ-algebra representing the knowledge of the underlying probability distribution available at time i. Martingales are required to satisfy three axioms for all i:

1. $\mathcal{F}_i \subseteq \mathcal{F}_{i+1}$. (The past is never forgotten.)
2. S_i is $\mathcal{F}_i$-measurable. (The present is always known.)
3. $\mathrm{E}[S_{i+1} \mid \mathcal{F}_i] = S_i$. (The future cannot be foreseen.)

Often, $\mathcal{F}_i$ will simply be the σ-algebra $\langle S_1, \ldots, S_i \rangle$ generated by the variables S_1 through S_i; in this case, axioms 1 and 2 will hold automatically.

To avoid special cases, let $\mathcal{F}_0$ denote the trivial σ-algebra consisting of the empty set and the entire probability space. The *difference sequence* of a martingale is the sequence $X_1, X_2, \ldots, X_n$, where $X_1 = S_1$ and $X_i = S_i - S_{i-1}$ for $i > 1$. A *zero-mean martingale* is a martingale for which $\mathrm{E}[S_i] = 0$.

4.3. Gambling systems. A remarkably useful theorem, which has its origins in the study of gambling systems, is due to Halmos [18]. We restate his theorem below in modern notation.

THEOREM 4.1. *Let $\{S_i, \mathcal{F}_i\}, 1 \le i \le n$, be a martingale with difference sequence $\{X_i\}$. Let $\{\zeta_i\}, 1 \le i \le n$, be random variables taking on the values* 0 *and* 1 *such that each ζ_i is $\mathcal{F}_{i-1}$-measurable. Then the sequence of random variables $S'_i = \sum_{j=1}^{i} \zeta_j X_j$ is a martingale relative to $\mathcal{F}_i$. (That is, $\{S'_i, \mathcal{F}_i\}$ is a martingale.)*

Proof. The first two properties are easily verified. Because ζ_i is $\mathcal{F}_{i-1}$-measurable, $\mathrm{E}[\zeta_i X_i \mid \mathcal{F}_{i-1}] = \zeta_i \mathrm{E}[X_i \mid \mathcal{F}_{i-1}] = 0$, and the third property also follows. □

4.4. Limit theorems. Many results that hold for sums of independent random variables carry over in modified form to martingales. For example, the following theorem of Hall and Heyde [17, Thm. 3.9] is a martingale version of the classical central limit theorem.

THEOREM 4.2 (see [17]). *Let $\{S_i, \mathcal{F}_i\}$ be a zero-mean martingale. Let $V_n^2 = \sum_{i=1}^{n} \mathrm{E}\left[X_i^2 \mid \mathcal{F}_{i-1}\right]$ and let $0 < \delta \le 1$. Define $L_n = \sum_{i=1}^{n} \mathrm{E}\left[|X_i|^{2+2\delta}\right] + \mathrm{E}\left[|V_n^2 - 1|^{1+\delta}\right]$.*

Then there exists a constant C depending only on δ such that whenever $L_n \leq 1$,

$$|\Pr[S_n \leq x] - \Phi(x)| \leq CL_n^{1/(3+2\delta)} \left[\frac{1}{1+|x|^{4(1+\delta)^2/(3+2\delta)}} \right], \tag{1}$$

where Φ is the standard unit normal distribution with mean 0 and variance 1.

For our purposes we will need only the case where x and δ are both set to 1. This allows the statement of the theorem to be simplified considerably. Furthermore, the rather complicated fraction containing x is never more than 1 and so can disappear into the constant. The result is:

THEOREM 4.3. *Let $\{S_i, \mathcal{F}_i\}$ be a zero-mean martingale. Let $V_n^2 = \sum_{i=1}^n E[X_i^2 \mid \mathcal{F}_{i-1}]$. Define $L_n = \sum_{i=1}^n \mathrm{E}\left[|X_i|^4\right] + \mathrm{E}\left[|V_n^2 - 1|^2\right]$. Then there exists a constant C such that whenever $L_n \leq 1$,*

$$|\Pr[S_n \leq 1] - \Phi(1)| \leq CL_n^{1/5}, \tag{2}$$

where Φ is the standard unit normal distribution with mean 0 and variance 1.

If we are interested only in the tails of the distribution of S_n, we can get a tighter bound using Azuma's inequality, a martingale analogue of the standard Chernoff bound [9] for sums of independent random variables. The usual form of this bound (see [2, 25]) assumes that the difference variables X_i satisfy $|X_i| \leq 1$. This restriction is too severe for our purposes, so below we prove a generalization of the inequality. In order to do so, we will need the following technical lemma.

LEMMA 4.4. *Let $\{S_i, \mathcal{F}_i\}, 1 \leq i \leq n$, be a zero-mean martingale with difference sequence $\{X_i\}$. Let $\mathcal{F}_0 \subseteq \mathcal{F}_1$ be a (not necessarily trivial) σ-algebra such that $\mathrm{E}[S_1 \mid \mathcal{F}_0] = 0$. If there exists a sequence of random variables $w_1, w_2, \ldots, w_n$ and a random variable W such that*

1. *W is $\mathcal{F}_0$-measurable,*
2. *each w_i is $\mathcal{F}_{i-1}$-measurable,*
3. *for all i, $|X_i| \leq w_i$ with probability 1, and*
4. *$\sum_{i=1}^n w_i^2 \leq W$ with probability 1,*

then for any $\alpha > 0$,

$$\mathrm{E}\left[e^{\alpha S_n} \mid \mathcal{F}_0\right] \leq e^{\alpha^2 W/2}. \tag{3}$$

Proof. The proof is by induction on n. First, notice that since $e^{\alpha X_1}$ is convex, we have

$$e^{\alpha X_1} \leq \left(\frac{w_1 - X_1}{2w_1}\right) e^{-\alpha w_1} + \left(1 - \frac{w_1 - X_1}{2w_1}\right) e^{\alpha w_1},$$

and thus

$$\begin{aligned} \mathrm{E}\left[e^{\alpha X_1} \mid \mathcal{F}_0\right] &\leq \mathrm{E}\left[\left(\frac{w_1 - X_1}{2w_1}\right) e^{-\alpha w_1} + \left(1 - \frac{w_1 - X_1}{2w_1}\right) e^{\alpha w_1} \mid \mathcal{F}_0\right] \\ &= \frac{1}{2} e^{-\alpha w_1} + \frac{1}{2} e^{\alpha w_1} - \left(\frac{e^{-\alpha w_1} - e^{\alpha w_1}}{2w_1}\right) \mathrm{E}[X_1 \mid \mathcal{F}_0] \\ &= \frac{1}{2} e^{-\alpha w_1} + \frac{1}{2} e^{\alpha w_1} \end{aligned}$$

since $\mathrm{E}[X_1 \mid \mathcal{F}_0]$ is zero.

Then, however,

$$\mathrm{E}\left[e^{\alpha X_1} \mid \mathcal{F}_0\right] \leq \frac{1}{2}\left(e^{-\alpha w_1} + e^{\alpha w_1}\right) = \cosh \alpha w_1 \leq e^{\alpha^2 w_1^2/2}.$$

If $n = 1$, we are done since $w_1^2 \leq W$. If n is greater than 1, for each $i \leq n-1$, let $S'_i = S_{i+1} - X_1$ and $\mathcal{F}'_i = \mathcal{F}_{i+1}$. Then $\{S'_i, \mathcal{F}'_i\}, 1 \leq i \leq n-1$, satisfies the conditions of the lemma with $\mathcal{F}'_0 = \mathcal{F}_1$, $w'_i = w_{i+1}$, and $W' = W - w_1^2$, so by the induction hypothesis, $E[e^{\alpha S'_{n-1}} \mid \mathcal{F}'_0] \leq e^{\alpha^2(W-w_1^2)/2}$. But then, using the fact that $\mathrm{E}[X \mid \mathcal{F}] = \mathrm{E}[\mathrm{E}[X \mid \mathcal{F}'] \mid \mathcal{F}]$ when $\mathcal{F} \subseteq \mathcal{F}'$, we can compute

$$\begin{aligned}
\mathrm{E}\left[e^{\alpha S_n} \mid \mathcal{F}_0\right] &= \mathrm{E}\left[\mathrm{E}\left[e^{\alpha X_1} e^{\alpha(S_n - X_1)} \mid \mathcal{F}_1\right] \mid \mathcal{F}_0\right] \\
&= \mathrm{E}\left[e^{\alpha X_1}\, \mathrm{E}\left[e^{\alpha S'_{n-1}} \mid \mathcal{F}'_0\right] \mid \mathcal{F}_0\right] \\
&\leq \mathrm{E}\left[e^{\alpha X_1} e^{\alpha^2(W-w_1^2)/2} \mid \mathcal{F}_0\right] \\
&= e^{\alpha^2(W-w_1^2)/2}\, \mathrm{E}\left[e^{\alpha X_1} \mid \mathcal{F}_0\right] \\
&\leq e^{\alpha^2(W-w_1^2)/2} e^{\alpha^2 w_1^2/2} \\
&= e^{\alpha^2 W/2}. \qquad \square
\end{aligned}$$

THEOREM 4.5. *Let $\{S_i, \mathcal{F}_i\}, 1 \leq i \leq n$, be a zero-mean martingale with difference sequence $\{X_i\}$. If there exists a sequence of random variables $w_1, w_2, \ldots, w_n$ and a constant W such that*

1. *each w_i is $\mathcal{F}_{i-1}$-measurable,*
2. *for all i, $|X_i| \leq w_i$ with probability 1, and*
3. *$\sum_{i=1}^n w_i^2 \leq W$ with probability 1,*

then for any $\lambda > 0$,

$$\Pr[S_n \geq \lambda] \leq e^{-\lambda^2/2W}. \tag{4}$$

Proof. By Lemma 4.4, for any $\alpha > 0$, $\mathrm{E}\left[e^{\alpha S_n}\right] \leq e^{\alpha^2 W/2}$. Thus by Markov's inequality,

$$\Pr[S_n \geq \lambda] = \Pr\left[e^{\alpha S_n} \geq e^{\alpha\lambda}\right] \leq e^{\alpha^2 W/2} e^{-\alpha\lambda}.$$

Setting $\alpha = \lambda/W$ gives (4). □

Symmetry immediately gives us the following corollary.

COROLLARY 4.6. *For any martingale $\{S_i, \mathcal{F}_i\}$ satisfying the premises of Theorem 4.5 and any $\lambda > 0$,*

$$\Pr[S_n \leq -\lambda] \leq e^{-\lambda^2/2W}. \tag{5}$$

Proof. Replace each S_i by $-S_i$ and apply Theorem 4.5. □

5. Proof of correctness. For this section, we will fix a particular scheduler. We may assume without loss of generality that the scheduler is deterministic because any random inputs the scheduler might use cannot depend on the history of the execution and therefore may also be fixed in advance.

Consider the sequence of random variables $X_1, X_2, \ldots$, where X_i represents (a) the ith vote that is decided by some processor executing line 7 or (b) 0 if fewer than i local coin flips occur. Note that the notion of the ith vote is well defined since we model concurrency by interleaving; it is assumed that the scheduler advances processors one at a time. For each i, let $\mathcal{F}_i$ be $\langle X_1, \ldots, X_i \rangle$, the σ-algebra generated by X_1 through X_i. Because the scheduler is deterministic, all of the random events in the system preceding the ith vote are captured in the variables X_1 through X_{i-1}, and the σ-algebra $\mathcal{F}_{i-1}$ thus determines the entire history of the system up to but not including the ith vote. Furthermore, since the scheduler's behavior depends only on the history of the system, $\mathcal{F}_{i-1}$ in fact determines the scheduler's choice of which processor will cast the ith vote. Thus conditioned on $\mathcal{F}_{i-1}$, X_i is just a random variable which takes on the values $\pm w$ with equal probability for some weight w determined by the scheduler's choice of which processor to run. Hence $\mathrm{E}[X_i \mid \mathcal{F}_{i-1}] = 0$, and the sequence of partial sums $S_i = \sum_{j=1}^{i} X_i$ is a martingale relative to $\{\mathcal{F}_i\}$.

We are not going to analyze $\{S_i, \mathcal{F}_i\}$ directly. Instead, it will be used as a base on which other martingales will be built using Theorem 4.1.

Let $\kappa_i = 1$ if $\sum_{j=1}^{i} X_j^2 \leq K$ and 0 otherwise. Votes for which $\kappa_i = 1$ will be called *common votes.* For each processor P, let $\zeta_i^P = 1$ if the vote X_i occurs before P reads, during its final read in line 13, the register of the processor that determines the value of X_i, and let $\zeta_i^P = 0$ otherwise. In effect, ζ_i^P is the indicator variable for whether P would see X_i if it were written out immediately. Observe that for a fixed scheduler, the values of both κ_i and ζ_i^P can be determined by examining the history of the system up to but not including the time when the vote X_i is cast, and thus both κ_i and ζ_i^P are $\mathcal{F}_{i-1}$-measurable. Consequently, the sequences $\{\sum_{j=1}^{i} \kappa_j X_j\}$ and $\{\sum_{j=1}^{i} \zeta_j^P X_j\}$ are martingales relative to $\{\mathcal{F}_i\}$ by Theorem 4.1. Votes for which $\zeta_i^P = 1$ but $\kappa_i = 0$ will be referred to as the *extra votes* for processor P. (Observe that $\zeta_i^P \geq \kappa_i$ since P could not have started its final read until the total variance was at least K.) The sequence $\{\sum_{j=1}^{i} (\zeta_i^P - \kappa_i) X_i\}$ of the partial sums of these extra votes is a difference of martingales and is thus also a martingale relative to $\{\mathcal{F}_i\}$.

The structure of the proof of correctness is as follows. First, we observe that the distribution of the total common vote, $\sum \kappa_i X_i$, is close to a normal distribution with mean 0 and variance K for suitable choices of a and K; in particular, we show that for n sufficiently large, the probability that $\sum \kappa_i X_i > \sqrt{K}$ will be at least a constant. Next, we complete the proof by showing that if the total common vote is far from the origin, the chances that any processor will read a total vote whose sign differs from the common vote is small. This fact is itself shown in two steps. First, it is shown that, for suitable choice of c, the total of the extra votes for a processor P, $\sum (\zeta_i^P - \kappa_i) X_i$, will be small with high probability. Second, a bound Δ is derived on the difference between $\sum \zeta_i^P X_i$ and the total vote actually read by P.

It will be necessary to select values for a, K, and c that give the correct bounds on the probabilities. However, we will be in a better position to justify our choice for these parameters after we have developed more of the analysis, so the choice of parameters will be deferred until §5.5.

5.1. Phases of the protocol. We begin by defining the phases of the protocol more carefully. Let t_i be the value of the ith processor's internal variable t at any given step of the protocol. Let U_i be the random variable representing the maximum value of t_i during the entire execution of the protocol. Let T_i be the random variable representing the maximum value of t_i during the part of the execution of the protocol where $\kappa = 1$.

In the proof of correctness, we will encounter many quantities of the form $\sum_{i=1}^{n} \chi(T_i)$ or $\sum_{i=1}^{n} \chi(U_i)$ for various functions χ. We will want to get bounds on these quantities without having to look too closely at the particular values of each T_i or U_i. This section proves several very general inequalities about quantities of this form, all of which are ultimately based on the following constraint:

$$K \geq \sum_i \sum_{j=1}^{T_i} j^{2a} \geq \sum_i \int_0^{T_i} j^{2a}\, dj = \sum_i \frac{{T_i}^{2a+1}}{2a+1}. \tag{6}$$

The constant $2a+1$ will reappear often; for convenience, we will write it as A. As noted above, $a \geq 0$, and hence $A \geq 1$.

Define $T_K = (AK/n)^{1/A}$ so that $K = nT_K^A/A$. The constant T_K represents the maximum value of each T_i if they are set to be equal while satisfying inequality (6). Note that T_K need not be an integer. Now we can show the following.

LEMMA 5.1. *Let $\psi(x) = x^A/A$ and let χ be any strictly increasing function such that $\chi\psi^{-1}$ is concave. Then for any nonnegative $\{x_i\}$, if $\sum_{i=1}^n \psi(x_i) \leq K$, then $\sum_{i=1}^n \chi(x_i) \leq n\chi(T_K)$.*

Proof. Since $\chi\psi^{-1}$ is concave, we have

$$\chi^{-1}\left(\sum \frac{\chi(x_i)}{n}\right) \leq \psi^{-1}\left(\sum \frac{\psi(x_i)}{n}\right)$$

[19, Thm. 92]. Simple algebraic manipulation yields

$$\sum \chi(x_i) \leq n\chi\left(\psi^{-1}\left(\sum \frac{\psi(x_i)}{n}\right)\right).$$

However,

$$\psi^{-1}\left(\sum \frac{\psi(x_i)}{n}\right) = \psi^{-1}\left(\frac{1}{n}\sum \frac{{x_i}^A}{A}\right) \leq \psi^{-1}\left(\frac{K}{n}\right) = T_K.$$

Hence $\sum \chi(x_i) \leq n\chi(T_K)$. □

Letting χ be the identity function, we have $\chi\psi^{-1}(x) = (Ax)^{1/A}$, which is concave for $A \geq 1$. Hence we have the following corollary.

COROLLARY 5.2.

$$\sum_{i=1}^n T_i \leq nT_K. \tag{7}$$

In the case where $\chi\psi^{-1}$ is convex, the following lemma applies instead.

LEMMA 5.3. *Let $\psi(x) = x^A/A$ and let χ be any strictly increasing function such that $\chi\psi^{-1}$ is convex. Then for any nonnegative $\{x_i\}$, if $\sum_{i=1}^n \psi(x_i) \leq K$, then $\sum_{i=1}^n \chi(x_i) \leq (n-1)\chi(0) + \chi(n^{1/A}T_K)$.*

Proof. Let $Y = \sum \psi(x_i)$. Now $\chi(x_i) = \chi\psi^{-1}\psi(x_i)$ or

$$\chi\psi^{-1}\left(\left(1 - \frac{\psi(x_i)}{Y}\right)0 + \frac{\psi(x_i)}{Y}Y\right),$$

which is at most

$$\left(1 - \frac{\psi(x_i)}{Y}\right)\chi\psi^{-1}(0) + \frac{\psi(x_i)}{Y}\chi\psi^{-1}(Y)$$

given the convexity of $\chi\psi^{-1}$. Hence

$$\begin{aligned}\sum_{i=1}^{n}\chi(x_i) &\leq n\chi\psi^{-1}(0) - \left(\sum_{i=1}^{n}\frac{\psi(x_i)}{Y}\right)\chi\psi^{-1}(0) + \left(\sum_{i=1}^{n}\frac{\psi(x_i)}{Y}\right)\chi\psi^{-1}(Y)\\ &= (n-1)\chi\psi^{-1}(0) + \chi\psi^{-1}\left(\sum_{i=1}^{n}\psi(x_i)\right)\\ &\leq (n-1)\chi\psi^{-1}(0) + \chi\psi^{-1}(K),\end{aligned}$$

which is just $(n-1)\chi(0) + \chi\left(n^{1/A}T_K\right)$. □

The quantity $n^{1/A}T_K$ is the maximum value that any x_i can take on without violating the constraint on $\sum x_i$. So what Lemma 5.3 says is that if $\chi\psi^{-1}$ is convex, $\sum \chi(x_i)$ is maximized by maximizing one of the x_i's while setting the rest to zero.

For the variables U_i, we can show the following.

LEMMA 5.4. *Let $\psi(x) = x^A/A$ and let χ be any strictly increasing function such that $\chi(\psi^{-1}(x) + c + 1)$ is concave in x. Then*

$$\sum_{i=1}^{n}\chi(U_i) \leq n\chi(T_K + c + 1). \tag{8}$$

Proof. Let W_i be the number of votes *written* to the registers during the part of the execution where the total of the register-variance fields is less than or equal to K. The set of variables $\{W_i\}$ satisfies the inequality $\sum W_i^A/A \leq K$ using the same argument as gives (6). Furthermore, $U_i \leq W_i + 1 + c$ because after the ith processor's next vote, the total variance in the registers must exceed K and it can cast at most c more votes before noticing this fact. Define $\chi'(x) = \chi(x + c + 1)$. Then $\chi(U_i) \leq \chi(W_i + c + 1) = \chi'(W_i)$. But ψ, χ', and W_i satisfy the premises of Lemma 5.1 and thus $\sum_{i=1}^{n}\chi(U_i) \leq \sum_{i=1}^{n}\chi'(W_i) \leq n\chi'(T_K) = n\chi(T_K + c + 1)$. □

Setting χ to be the identity function gives the following result.

COROLLARY 5.5.

$$\sum_{i=1}^{n} U_i \leq n(T_K + c + 1). \tag{9}$$

Proof. $\chi(\psi^{-1}(x) + c + 1) = Ax^{1/A} + c + 1$, which is concave since $A \geq 1$. □

Define $g = 1 + (c+3)/T_K$; then $gT_K = T_K + c + 3$ will be an upper bound for $T_K + c + 1$ as well as a number of closely related constants involving c that will appear later.

5.2. Common votes.

The purpose of this section is to show that for n sufficiently large, the total common vote is far from the origin with constant probability. We do so by showing that under the right conditions, the total common vote will be nearly normally distributed.

Let $S_i^K = \sum_{j=1}^{i}\kappa_j X_j$. As pointed out above, $\{S_i^K = \sum_{j=1}^{i}\kappa_j X_j, \mathcal{F}_i\}$ is a martingale. Let $N = \lceil nT_K \rceil$. It follows from Corollary 5.2 that $\kappa_i = 0$ for $i > N$ and thus $S_N^K = \lim_{i\to\infty} S_i^K$ is the sum of all the common votes. The distribution of S_N^K is characterized in the following lemma.

LEMMA 5.6. *If*

$$\frac{4A^2}{n^{1/A}T_K} \leq 1, \tag{10}$$

then

$$\left|\Pr\left[S_N^K \le \sqrt{K}\right] - \Phi(1)\right| \le C_1 \left(\frac{A^2}{n^{1/A} T_K}\right)^{1/5}, \tag{11}$$

where C_1 is an absolute constant.

Proof. The proof uses Theorem 4.3, which requires that the martingale be normalized so that the total conditional variance V_N^2 is close to 1. So let $Y_i = \kappa_i X_i/\sqrt{K}$ and consider the martingale $\{\sum_{j=1}^i Y_j, \mathcal{F}_i\}$. To apply the theorem, we need to compute a bound on the value L_N.

We begin by getting a bound on the first term $\sum \mathrm{E}\left[|Y_i|^4\right]$. We have

$$\sum_{i=1}^N \mathrm{E}\left[|Y_i|^4\right] = \mathrm{E}\left[\sum_{i=1}^N |Y_i|^4\right] = \frac{1}{K^2}\mathrm{E}\left[\sum_{i=1}^N |\kappa_i X_i|^4\right] = \frac{1}{K^2}\mathrm{E}\left[\sum_{i=1}^n \sum_{j=1}^{T_i} j^{4a}\right]. \tag{12}$$

Now

$$\sum_{j=1}^{T_i} j^{4a} \le \int_0^{T_i} j^{4a}\,dj + T_i^{4a} = \frac{T_i^{4a+1}}{4a+1} + T_i^{4a}.$$

Consider the two parts of this bound separately. Define $\psi(x) = x^A/A, \chi(x) = x^{4a+1}/(4a+1)$; then $\chi\psi^{-1}(y) = (Ay)^{(4a+1)/A}/(4a+1)$ is convex, $\chi(0) = 0$, and hence $\sum_{i=1}^n T_i^{4a+1}/(4a+1)$ is at most $(n^{1/A}T_K)^{4a+1}/(4a+1)$ using Lemma 5.3.

Similarly, let $\chi'(x) = x^{4a}$. Here the convexity of $\chi'\psi^{-1}$ depends on the value of a. If $a \ge 1/2$, then $\chi'\psi^{-1}(y) = (Ay)^{4a/A}$ is convex (since $4a/A = 4a/(2a+1) \ge 1$), and thus (again by Lemma 5.3) $\sum_{i=1}^n T_i^{4a} \le (n^{1/A}T_K)^{4a} = n^{4a/A}T_K^{4a} \le n^{(4a+1)/A}T_K^{4a}$. If $a \le 1/2$, then $\chi'\psi^{-1}(y)$ is concave (since now $4a/A \le 1$), and thus by Lemma 5.1, $\sum_{i=1}^n T_i^{4a} \le nT_K^{4a} \le n^{(4a+1/A)}T_K^{4a}$.

Plugging everything back into (12) gives

$$\sum_{i=1}^N \mathrm{E}\left[|Y_i|^4\right] \le \frac{n^{(4a+1)/A}T_K^{4a}}{K^2} + \frac{(n^{1/A}T_K)^{4a+1}}{K^2(4a+1)}. \tag{13}$$

For the second term $\mathrm{E}\left[|V_N^2 - 1|^2\right]$, observe that

$$V_N^2 = \sum_{i=1}^N \mathrm{E}\left[Y_i^2 \mid \mathcal{F}_{i-1}\right] = \frac{1}{K}\sum_{i=1}^N \mathrm{E}\left[(\kappa_i X_i)^2 \mid \mathcal{F}_{i-1}\right],$$

which is just $1/K$ times the sum of the squares of the weights $|X_i|$ of the common votes. But the total variance of the common votes can differ from K by at most the variance of the first vote X_i for which $\kappa_i = 0$. Since the processor that casts this vote can have cast at most $n^{1/A}T_K$ votes beforehand, the variance of this vote is at most $(n^{1/A}T_K + 1)^{2a}$, giving the bound

$$|V_N^2 - 1|^2 \le \frac{1}{K^2}\left(n^{1/A}T_K + 1\right)^{4a}. \tag{14}$$

Combining (13) and (14) gives

$$L_N \le \frac{n^{(4a+1)/A}T_K^{4a}}{K^2} + \frac{(n^{1/A}T_K)^{4a+1}}{K^2(4a+1)} + \frac{\left(n^{1/A}T_K + 1\right)^{4a}}{K^2}$$

$$\begin{aligned}
&= \frac{n^{(4a+1)/A}T_K^{4a}}{K^2} + \frac{n^{(4a+1)/A}T_K^{4a+1}}{K^2(4a+1)} + \frac{n^{4a/A}T_K^{4a}(1+n^{-1/A}T_K^{-1})^{4a}}{K^2} \\
&\le A^2 n^{-1/A}T_K^{-2} + \frac{A^2 n^{-1/A}T_K^{-1}}{4a+1} + A^2 n^{-2/A}T_K^{-2}\exp(4an^{-1/A}T_K^{-1}) \\
&\le 2A^2 n^{-1/A}T_K^{-1} + e^{1/2}A^2 n^{-1/A}T_K^{-1} \\
&< \frac{4A^2}{n^{1/A}T_K}.
\end{aligned}$$

The third-to-last step uses the approximation $(1+x)^b \le e^{bx}$ for nonnegative b and x. The resulting exponential term is serendipitously bounded by $e^{1/2}$ if (10) holds since $2a < A \le A^2$ implies $4an^{-1/A}T_K^{-1} < 2A^2(n^{1/A}T_K)^{-1} \le 2/4$.

A more direct application of (10) shows that $L_N \le 1$, and thus Theorem 4.3 applies. Hence

$$\begin{aligned}
\left|\Pr\left[\sum \kappa_i X_i \le \sqrt{K}\right] - \Phi(1)\right| &= \left|\Pr\left[\sum_{i=1}^{N} Y_i \le 1\right] - \Phi(1)\right| \\
&\le C\left(\frac{4A^2}{n^{1/A}T_K}\right)^{1/5} \\
&\le C_1\left(\frac{A^2}{n^{1/A}T_K}\right)^{1/5}. \qquad \square
\end{aligned}$$

5.3. Extra votes. In this section, we examine the extra votes from the point of view of a particular processor P.

Recall that ζ_i^P is defined to be 1 if the vote X_i is cast by some processor Q before P's final read of Q's register and 0 otherwise. Clearly, $\zeta_i^P \ge \kappa_i$ since P could not have started its final read until the total variance exceeded K. As discussed above, both ζ_i^P and κ_i are $\mathcal{F}_{i-1}$-measurable. Thus $\xi_i = \zeta_i^P - \kappa_i$ is a $0-1$ random variable that is $\mathcal{F}_{i-1}$-measurable, and $\{S_i^P = \sum_{j=1}^{i} \xi_j X_j, \mathcal{F}_i\}$ is a martingale by Theorem 4.1.

Define $\Delta = n(gT_K)^a$. The following lemma shows a bound on the tails of $\sum \xi_i X_i$.

LEMMA 5.7. *If*

$$g^a \le \frac{1}{2}\sqrt{\frac{T_K}{nA}} \tag{15}$$

and

$$g^A \le 1 + \frac{1}{8\log(10n)},^{7} \tag{16}$$

then for each processor P,

$$\Pr\left[\sum(\zeta_i^P - \kappa_i)X_i \le \Delta - \sqrt{K}\right] \le \frac{1}{10n}. \tag{17}$$

Proof. The proof uses Corollary 4.6, so we proceed by showing that its premises (stated in Theorem 4.5) are satisfied for $\{\sum \xi_i X_i, \mathcal{F}_i\}$.

By Corollary 5.5, X_i and thus $\xi_i X_i$ is zero for $i > n(T_K + c + 1)$. Thus $\sum \xi_i X_i = S_M^P$, where $M = n(T_K + c + 1)$.

[7] By $\log(x)$, we will always mean the natural logarithm of x.

Set $w_i = |\xi_i X_i|$. Then the first premise of Corollary 4.6 follows from the fact that for each i, ξ_i and $|X_i|$ are both $\mathcal{F}_i$-measurable. The second premise is immediate. For the third premise, notice that

$$\sum(|\xi_i X_i|)^2 = \sum \xi_i X_i^2 = \sum \zeta_i^P X_i^2 - \sum \kappa_i X_i^2 \le \sum X_i^2 - \sum \kappa_i X_i^2.$$

The first term is

$$\sum X_i^2 = \sum_{i=1}^{n} \sum_{j=1}^{U_i} j^{2a}.$$

The second term is

$$\sum \kappa_i X_i^2 \ge K - t^{2a}$$

for some t which is at most U_i for some i. Thus

$$\begin{aligned}
\sum(|\xi_i X_i|)^2 &\le -K + t^{2a} + \sum_{i=1}^{n} \sum_{j=1}^{U_i} j^{2a} \\
&< -K + \sum_{i=1}^{n} \sum_{j=1}^{U_i+1} j^{2a} \\
&\le -K + \sum_{i=1}^{n} (U_i + 2)^A / A. \qquad (18)
\end{aligned}$$

Let $\chi(x) = (x+2)^A/A$. Then

$$\chi\left(\psi^{-1}(y) + c + 1\right) = \frac{\left((Ay)^{1/A} + c + 3\right)^A}{A}. \qquad (19)$$

We can treat this function as an instance of a class of functions of the form $(x^p + C)^q$, where x, p, q, and C are all nonnegative, whose concavity (or lack thereof) can be determined by finding the sign of the second derivative:

$$\begin{aligned}
\operatorname{sgn}\left[\frac{d^2}{dx^2}(x^p + C)^q\right] &= \operatorname{sgn}\left[\frac{d}{dx} q(x^p + C)^{q-1} p x^{p-1}\right] \\
&= \operatorname{sgn}\left[q(q-1)(x^p + C)^{q-2} p^2 x^{2p-2} + q(x^p + C)^{q-1} p(p-1) x^{p-2}\right] \\
&= \operatorname{sgn}\left[q(x^p + C)^{q-2} p x^{p-2}\left[(q-1) p x^p + (x^p + C)(p-1)\right]\right] \\
&= \operatorname{sgn}\left[(q-1) p x^p + (x^p + C)(p-1)\right] \\
&= \operatorname{sgn}\left[(pq-1) x^p + C(p-1)\right].
\end{aligned}$$

In the particular case we are interested in, $p = 1/A$, $q = A$, and $C = c + 3$. Since $pq - 1 = 0$, the first term vanishes and the sign is equal to the sign of $1/A - 1$, which is less than or equal to zero since $A \ge 1$. Thus the function $(x^{1/A} + c + 3)^A$ is concave, and since concavity is preserved by linear transformations, $((Ay)^{1/A} + c + 3)^A/A$ is concave as well.

Lemma 5.4 now gives

$$\sum_{i=1}^{n} \frac{(U_i + 2)^A}{A} \le n\chi(T_K + c + 1) = \frac{n(T_K + c + 3)^A}{A} \le \frac{n(gT_K)^A}{A}. \qquad (20)$$

It follows from (18) and (20) that

$$\sum(|\xi_i X_i|)^2 \leq \frac{n(gT_K)^A}{A} - K = K(g^A - 1).$$

Applying (5) from Corollary 4.6 now yields, for all $\lambda > 0$,

$$\Pr\left[S_M^P \leq -\lambda\right] \leq e^{-\lambda^2/(2K(g^A-1))}. \tag{21}$$

If (15) holds, then $\Delta \leq \sqrt{K}/2$. Thus

$$\begin{aligned}\Pr\left[\sum \xi_i X_i \leq \Delta - \sqrt{K}\right] &\leq \Pr\left[S_M^P \leq -\frac{\sqrt{K}}{2}\right] \\ &\leq e^{-K/(8K(g^A-1))} \\ &= e^{-1/(8(g^A-1))}.\end{aligned}$$

But if (16) holds, then

$$g^A - 1 \leq \frac{1}{8\log(10n)}$$

and, since $\log(10n) > 0$ and $g > 1$,

$$-\frac{1}{8(g^A-1)} \leq -\log(10n),$$

from which it follows that

$$e^{-1/8(g^A-1)} \leq e^{-\log(10n)} = \frac{1}{10n}. \qquad \square$$

5.4. Written votes vs. decided votes. In this section, we show that the difference between $\sum \zeta_i^P X_i$ and the total vote actually read by P is bounded by $\Delta = n(gT_K)^a$.

LEMMA 5.8. *Let R_P be the sum of the votes read during P's final read. Then*

$$\left|\sum \zeta_i^P X_i - R_P\right| \leq n(T_k + c + 1)^a \leq n(gT_K)^a = \Delta. \tag{22}$$

Proof. Suppose $\zeta_i^P = 1$, and suppose X_i is decided by processor P_j. If the vote X_i is not included in the value read by P, it must have been decided before P's read of P_j's register but written afterwards. Because each vote is written out before the next vote is decided, there can be at most one vote from P_j which is included in $\sum \zeta_i^P X_i$ but is not actually read by P. This vote has weight at most U_j^a. Thus we have $|\sum \zeta_i^P X_i - R_P| \leq \sum_{i=1}^n U_i^a$.

Now let $\chi(x) = x^a$. Then $\chi(\psi^{-1}(y) + c + 1) = ((Ay)^{1/A} + c + 1)^a$. The concavity of this function can be shown using the argument applied to (19) in Lemma 5.7: the sign of its second derivative will be equal to the sign of $(pq-1)x^p + C(p-1)$, where $x = Ay$, $p = 1/A$, $q = a$, and $C = c + 1$. Since Ay and $c+1$ are both nonnegative and a/A and $1/A$ are both less than or equal to 1, both terms are nonpositive and thus $((Ay)^{1/A} + c + 1)^a$ is concave. The rest follows from Lemma 5.4. $\square$

5.5. Choice of parameters. Let us summarize the proof of correctness in a single theorem.

THEOREM 5.9. *Define*

$$A = 2a + 1,$$
$$T_K = \left(\frac{AK}{n}\right)^{1/A},$$
$$g = 1 + \frac{c+3}{T_K}$$

and suppose that all of the following hold:

$$g^a \le \frac{1}{2}\sqrt{\frac{T_K}{nA}}, \tag{23}$$

$$g^A \le 1 + \frac{1}{8\log(10n)}, \tag{24}$$

$$\frac{4A^2}{n^{1/A}T_K} \le 1. \tag{25}$$

Then the protocol implements a shared coin with agreement parameter at least

$$1 - \left[\Phi(1) + C_1\left(\frac{A^2}{n^{1/A}T_K}\right)^{1/5} + \frac{1}{10}\right], \tag{26}$$

where C_1 is the constant from Lemma 5.6.

Proof. To show that the agreement parameter is at least (26), we must show that for each $z \in \{0,1\}$, the probability that all processors decide z is at least (26). Without loss of generality, let us consider only the probability that all processors decide 1; the case of all processors deciding 0 follows by symmetry.

The essential idea of the proof is as follows. With at least a constant probability, the total common vote is at least $\sqrt{K}$ (Lemma 5.6). The "drift" added to this total by the extra votes for any single processor P is small with high probability (Lemma 5.7). Thus even after adding in the extra votes for P, the total will be large enough that the offset $\Delta = n(gT_K)^a$ caused by votes that are generated but not written out in time for P's final read will not push it over the line (Lemma 5.8).

More formally, we wish to show that the event

- $\sum \kappa_i X_i > \sqrt{K}$ and
- for each P, $\sum(\zeta_i^P - \kappa_i)X_i > \Delta - \sqrt{K}$

occurs with probability at least (26). Since this event implies that for all P, $\sum \zeta_i^P X_i > \Delta$, by Lemma 5.8, we have that each P reads a value greater than 0 during its final read and thus decides 1.

It will be easiest to compute an upper bound on the probability that this event does *not* occur. For the event not to occur, we must have either $\sum \kappa_i X_i \le \sqrt{K}$ or $\sum(\zeta_i^P - \kappa_i)X_i \le \Delta - \sqrt{K}$ for some P. But since the probability of a union of events never exceeds the sum of the probabilities of the events, the probability of failing in any of these ways is at most

$$\Pr\left[\sum \kappa_i X_i \le \sqrt{K}\right] + \sum_P \Pr\left[\sum(\zeta_i^P - \kappa_i)X_i \le \Delta - \sqrt{K}\right]$$

$$\le \left[\Phi(1) + C_1\left(\frac{A^2}{n^{1/A}T_K}\right)^{1/5}\right] + n\frac{1}{10n} \tag{27}$$

by Lemmas 5.6 and 5.7. Therefore, the probability that some processor decides 0 is at most (27), and thus the probability that all processors decide 1 is at least 1 minus (27). □

The running time of the protocol is more easily shown.

THEOREM 5.10. *No processor executes more than* $(AK)^{1/A}(2 + n/c) + 2c + 2n$ *register operations during an execution of the shared-coin protocol.*

Proof. First, consider the maximum number of votes a processor can cast. After $(AK)^{1/A}$ votes, the total variance of the processor's votes will be

$$\sum_{x=1}^{(AK)^{1/A}} x^{2a} > \int_0^{(AK)^{1/A}} x^{2a}\, dx = \frac{\left((AK)^{1/A}\right)^A}{A} = K,$$

so after at most an additional c votes, the processor will execute line 11 of Figure 1 and see a total variance greater than K. Thus each processor casts at most $(AK)^{1/A} + c$ votes. But each vote costs 1 write operation in line 8, and every c votes costs n reads in line 11, to which must be added a one-time cost of n reads in line 13. The total number of operations is thus at most $((AK)^{1/A} + c)\,(1 + \lceil n/c \rceil) + n \leq ((AK)^{1/A} + c)(2 + n/c) + n = (AK)^{1/A}(2 + n/c) + 2c + 2n$. □

It remains only to find values for a, K, and c which give both a constant agreement parameter and a reasonable running time. As a warm-up, let us consider what happens if we emulate the protocol of Bracha and Rachman [8].

THEOREM 5.11. *If* $a = 0$, $K = 4n^2$, *and* $c = n/(4 \log n) - 3$, *then for* n *sufficiently large, the protocol implements a shared coin with agreement parameter at least* 0.05 *in which each processor executes at most* $O(n^2 \log n)$ *operations.*

Proof. For the agreement parameter, we have $A = 1$, $T_K = 4n$, and $g = 1 + 1/(16 \log n)$. Then (23) holds since $g^a = 1 \leq (1/2)\sqrt{T_K/nA} = 1$. Furthermore,

$$\begin{aligned}\left(1 + \frac{1}{8 \log(10n)}\right)^{1/A} &= 1 + \frac{1}{8(\log n + \log 10)}\\ &\geq 1 + \frac{1}{16 \log n}\end{aligned}$$

when $n \geq 10$. Thus (24) holds. The remaining inequality (25) holds for $n \geq 1$, so by Theorem 5.9, we have a probability of failure of at most

$$\begin{aligned}&\Phi(1) + C_1 \left(\frac{1}{4n^2}\right)^{1/5} + \frac{1}{10}\\ &\leq 0.842 + O\left(\frac{1}{n^{2/5}}\right) + 0.1,\end{aligned}$$

which is not more than $0.942 + \epsilon$ for n sufficiently large. In particular, for n greater than some n_0, this quantity is at most 0.95, and the agreement parameter is thus at least $1 - 0.95$.

The running time is immediate from Theorem 5.10. □

Now consider what happens if a is not restricted to be a constant 0.

THEOREM 5.12. *If* $a = (\log n - 1)/2$, $K = (16 n \log n)^{\log n}(n/\log n)$, *and* $c = (n/\log n) - 3$, *then for* n *sufficiently large, the protocol implements a shared coin with constant agreement parameter in which each processor executes at most* $O(n \log^2 n)$ *operations.*

Proof. We have $A = \log n$, $T_K = 16n \log n$, and $g = 1 + 1/(16 \log^2 n)$.

We want to apply Theorem 5.9, so first we verify that its premises are satisfied. To show (23), compute

$$g^a = \left(1 + \frac{1}{16 \log^2 n}\right)^{(\log n - 1)/2} \leq e^{(\log n - 1)/(32 \log^2 n)} \leq e^{1/(32 \log n)},$$

which for $n \geq 2$ will be less than $(1/2)\sqrt{T_K/nA} = 2$. To show (24), note that

$$g^A = \left(1 + \frac{1}{16 \log^2 n}\right)^{\log n} \leq e^{1/(16 \log n)}$$

and thus $\log(g^A) \leq 1/(16 \log n)$. But

$$\begin{aligned} \log\left(1 + \frac{1}{8 \log(10n)}\right) &\geq \frac{1}{8 \log(10n)} - \frac{1}{128 \log^2(10n)} \\ &= \frac{1}{8(\log n + \log 10)} - \frac{1}{128(\log n + \log 10)^2} \end{aligned}$$

(using the approximation $\log(1 + x) \geq x - (1/2)x^2$). For sufficiently large n, this quantity exceeds $1/(16 \log n)$ and (24) holds. The remaining constraint (25) is easily verified, and thus Theorem 5.9 applies and the agreement parameter is at least

$$\begin{aligned} 1 - &\left[\Phi(1) + C_1 \left(\frac{\log^2 n}{n^{1/\log n}(16n \log n)}\right)^{1/5} + \frac{1}{10}\right] \\ &\geq 1 - \left(0.842 + O\left(\left(\frac{\log n}{n}\right)^{1/5}\right) + 0.10\right), \end{aligned}$$

which is at least 0.05 for sufficiently large n. Thus the protocol gives a constant agreement parameter.

Now by Theorem 5.10, the number of operations executed by any single processor is at most $(AK)^{1/A}(2 + n/c) + 2c + 2n$, or

$$(\log n)^{1/\log n}(16n \log n)(n/\log n)^{1/\log n} O(\log n) + O(n),$$

which is $O(n \log^2 n)$. □

6. Discussion. This paper presents the first randomized consensus algorithm which achieves a nearly optimal worst-case bound on the expected number of operations a processor needs to execute. To achieve this, we construct a weak-shared-coin protocol based on random voting where the weight of votes cast by a processor increases with the number of votes it has already cast. The consensus protocol can then be constructed around it using the established techniques of Aspnes and Herlihy [4] with only a constant-factor increase in the number of operations done by each processor.[8]

This work leads to several interesting questions. First, our voting scheme implicitly gives higher priority to operations done by processors that have already performed

[8] Due to the unbounded round structure of [4], the resulting consensus protocol assumes unbounded registers. We believe these unbounded registers can be eliminated using the bounded round numbers construction of Dwork, Herlihy, and Waarts [14].

many operations. Such implicit priority granting may yield faster algorithms for other shared-memory problems, such as approximate agreement or randomized resource allocation.

Also, although our solution improves significantly on the worst-case expected bound on the number of operations a single processor is required to perform in order to achieve consensus, the total number of operations done by all of the processors together is slightly larger (by a factor of $\log n$) than in the unweighted-voting protocol of Bracha and Rachman [8]. It is of theoretical interest whether there is an inherent trade-off here.

Acknowledgments. We would like to thank Serge Plotkin and David Applegate for their many useful suggestions.

REFERENCES

[1] K. Abrahamson, *On achieving consensus using a shared memory*, in Proc. 7th ACM SIGACT–SIGOPS Symposium on Principles of Distributed Computing, Association for Computing Machinery, New York, 1988, pp. 291–302.

[2] N. Alon and J. H. Spencer, *The Probabilistic Method*, John Wiley, New York, 1992.

[3] J. Aspnes, *Time- and space-efficient randomized consensus*, J. Algorithms, 14 (1993), pp. 414–431.

[4] J. Aspnes and M. Herlihy, *Fast randomized consensus using shared memory*, J. Algorithms, 11 (1990), pp. 441–461.

[5] H. Attiya, D. Dolev, and N. Shavit, *Bounded polynomial randomized consensus*, in Proc. 8th ACM Symposium on Principles of Distributed Computing, Association for Computing Machinery, New York, 1989, pp. 281–294.

[6] P. Billingsley, *Probability and Measure*, 2nd ed., John Wiley, New York, 1986.

[7] G. Bracha and O. Rachman, *Approximated counters and randomized consensus*, Tech. report 662, Technion, Haifa, Israel, 1990.

[8] ———, *Randomized consensus in expected $O(n^2 \log n)$ operations*, in Proc. 5th International Workshop on Distributed Algorithms, Springer-Verlag, Berlin, New York, Heidelberg, 1991.

[9] H. Chernoff, *A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations*, Ann. Math. Stat., 23 (1952), pp. 493–407.

[10] B. Chor, A. Israeli, and M. Li, *Wait-free consensus using asynchronous hardware*, SIAM J. Comput., 23 (1994), pp. 701–712; preliminary version appears in Proc. 6th ACM SIGACT–SIGOPS Symposium on Principles of Distributed Computing, Association for Computing Machinery, New York, pp. 86–97, 1987.

[11] B. Chor and L. Moscovici, *Solvability in asynchronous environments*, in Proc. 30th Annual IEEE Conference on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1989, pp. 422–427.

[12] D. Dolev, C. Dwork, and L. Stockmeyer, *On the minimal synchronism needed for distributed consensus*, J. Assoc. Comput. Mach., 34 (1987), pp. 77–97.

[13] C. Dwork, M. Herlihy, S. Plotkin, and O. Waarts, *Time-lapse snapshots*, in Proc. Israel Symposium on the Theory of Computing and Systems, Springer-Verlag, Berlin, New York, Heidelberg, 1992, pp. 159–170.

[14] C. Dwork, M. Herlihy, and O. Waarts, *Bounded round numbers*, in Proc. 12th ACM SIGACT–SIGOPS Symposium on Principles of Distributed Computing, Association for Computing Machinery, New York, 1993, pp. 53–64.

[15] W. Feller, *An Introduction to Probability Theory and Its Applications*, vol. 2, 2nd ed., John Wiley, New York, 1971.

[16] M. J. Fischer, N. A. Lynch, and M. S. Paterson, *Impossibility of distributed commit with one faulty process*, J. Assoc. Comput. Mach., 32 (1985), pp. 374–382.

[17] P. Hall and C. Heyde, *Martingale Limit Theory and Its Application*, Academic Press, New York, 1980.

[18] P. R. Halmos, *Invariants of certain stochastic transformations: The mathematical theory of gambling systems*, Duke Math. J., 5 (1939), pp. 461–478.

[19] G. Hardy, J. Littlewood, and G. Pólya, *Inequalities*, 2nd ed., Cambridge University Press, Cambridge, UK, 1952.

[20] M. Herlihy, *Wait-free synchronization*, ACM Trans. Programming Lang. Systems, 13 (1991), pp. 124–149.

[21] P. Kopp, *Martingales and Stochastic Integrals*, Cambridge University Press, Cambridge, UK, 1984.

[22] M. C. Loui and H. H. Abu-Amara, *Memory requirements for agreement among unreliable asynchronous processes*, in Advances in Computing Research, vol. 4, F. P. Preparata, ed., JAI Press, Greenwich, CT, 1987.

[23] S. A. Plotkin, *Sticky bits and universality of consensus*, in Proc. 8th ACM Symposium on Principles of Distributed Computing, Association for Computing Machinery, New York, 1989, pp. 159–176.

[24] M. Saks, N. Shavit, and H. Woll, *Optimal time randomized consensus: Making resilient algorithms fast in practice*, in Proc. 2nd Annual ACM–SIAM Symposium on Discrete Algorithms, Society for Industrial and Applied Mathematics, Philadelphia, 1991, pp. 351–362.

[25] J. Spencer, *Ten Lectures on the Probabilistic Method*, Society for Industrial and Applied Mathematics, Philadelphia, 1987.

SIAM J. COMPUT.
Vol. 25, No. 5, pp. 1045–1060, October 1996

006

OPTIMAL GROUP GOSSIPING IN HYPERCUBES UNDER A CIRCUIT-SWITCHING MODEL*

SATOSHI FUJITA† AND MASAFUMI YAMASHITA†

Abstract. Let U be a given set of nodes of a parallel computer system and assume that each node u in U has a piece of information $t(u)$ called a token. This paper discusses the problem of each $u \in U$ broadcasting its token $t(u)$ to all nodes in U. We refer to this problem as the group-gossiping problem, which includes the (conventional) gossiping problem as a special case. In this paper, we consider the group-gossiping problem in n-cubes under a circuit-switching model and propose an optimal group-gossiping algorithm for n-cubes under the model.

Key words. parallel algorithm, gossiping, circuit-switching model, optimal-time bound, n-cubes

AMS subject classifications. 05C38, 68Q20, 68R10

1. Introduction. Massively parallel computer systems usually consist of enormous number of processors (called nodes) connected by communication links and are characterized by the absence of shared memory. In them, nodes communicate with each other by message-passing via communication links. This paper discusses an information-dissemination problem that is a natural extension of the gossiping problem for parallel computer systems.

> Let V be the set of nodes in a parallel computer system, and let U be a subset of V. We assume that each node $u \in U$ has a piece of information $t(u)$ called a *token*. Then the *group-gossiping problem* for U is the problem of broadcasting the tokens to all nodes in U and making every node $u \in U$ know all tokens $t(v)$ $(v \in U)$.

When $U = V$, it is the conventional gossiping problem, which has been investigated extensively during the past decade [2, 3, 8, 12, 11, 14, 18, 22], and when $|U| = 2$, it is the point-to-point routing problem [4, 20].

By definition, any gossiping algorithm can correctly solve the group-gossiping problem since $U \subseteq V$. Suppose that a node can communicate a message to another node in a step (i.e., in one unit of time) only if there is a communication link between them. We call this the assumption of *link communication*. It captures well a characteristic of the *store-and-forward routing* communication scheme.[1] Assuming link communication, group gossiping requires time greater than or equal to the diameter of the network topology of the system in the worst case. On the other hand, for each of many network topologies, there has been proposed a gossiping algorithm which achieves a gossiping time close to the diameter of the network topology even if the links are half-duplex, i.e., bidirectional communication via a link in a time unit is

* Received by the editors July 28, 1993; accepted for publication (in revised form) January 25, 1995. Earlier versions of some results contained in this paper appear in the following paper: S. Fujita, M. Yamashita, and T. Ae, "Optimal group gossiping in hypercubes under wormhole routing model," in *Proc. 4th International Symposium on Algorithms and Computation,* Lecture Notes in Computer Science 762, Springer-Verlag, Berlin, 1993, pp. 227–286.

† Department of Electrical Engineering, Faculty of Engineering, Hiroshima University, Kagamiyama 1-4-1, Higashi-Hiroshima 739, Japan (fujita@csl.hiroshima-u.ac.jp, mak@se.hiroshima-u.ac.jp).

[1] The store-and-forward routing scheme sends messages as packets. Execution of a send (packet) instruction at a node picks a packet from its queue and sends it to the queue of the adjacent node specified in the packet through the link connecting the two nodes. The packet is delivered to its destination by repeating local packet transmission.

not permitted [19].[2] Therefore, the group-gossiping problem is efficiently solvable by elaborate gossiping algorithms for parallel computer systems adopting the store-and-forward routing communication scheme (i.e., when we assume link communication).

Circuit switching is an alternative communication scheme for parallel computer systems and telecommunication networks. In this routing scheme, a node wishing to send a message to a destination node first reserves a route from the source node to the destination node and then flows a message to the destination node along the reserved route in a pipelined manner [20]. It is usually assumed that a message of unit length sent by a node can rearch its destination node in unit time. We call this the assumption of *line communication.*[3] This paper investigates the group-gossiping problem for the circuit-switching routing communication scheme (i.e., assuming line communication).

An intuitive description of the model we have in mind is the following[4] (a formal description of the model will be given in §2):

> Suppose that a node u wishes to send a message M to a node v through route P. Then communication is delayed until all links in P become free. Once communication starts, it finishes by a small constant time τ, where τ is insensitive to nodes u and v, message M, and route P. If more than two communications occur simultaneously via routes sharing links, exactly one of them succeeds. In other words, we need to reserve the route before starting communication, but once it is reserved, communication is made quickly.

Although communication is made in a constant time regardless of the locations of the communicating nodes, it is required to reserve a route connecting them for communication. In order for k pairs (u_i, v_i) $(1 \leq i \leq k)$ of nodes to communicate with each other simultaneously, we must reserve k edge-disjoint routes connecting those pairs. A group-gossiping algorithm requires many nodes to send tokens in parallel to achieve short gossiping time, but the degree of possible parallelism is obviously bounded by, e.g., the edge-connectivity of the network topology, and, moreover, the given subset U may be distributed inadequately for parallel communication. The main theme of this paper is how to extract parallelism under those conditions. To simplify the explanation and to clarify the analysis, in this paper, we assume that the time required for reserving a route is negligible, although it in practice would be proportional to the length of the route to be reserved.

As the underlying network topology G, i.e., $G = (V, E)$, where V is the set of nodes and E is the set of links, we adopt n-cubes, which are the most popular topology for commercial parallel computer systems (e.g., iPSC/2 and nCUBE-2). We propose two group-gossiping algorithms for n-cubes. Let $|U| = 2^{\epsilon n}$ for $\epsilon \leq 1$. The first algorithm completes the group-gossiping in time $\lceil \log_2(|U|-1)/\log_2 n \rceil + o(\log_2(|U|-1)/\log_2 n)$, provided that $1/(1-\epsilon) = o(n)$, which asymptotically achieves a lower bound $\lceil \log_2(|U|-1)/\log_2 n \rceil$. The second algorithm, on the other hand, achieves the

[2] If we further assume that each node can communicate with at most one neighbor node in each time unit, in many cases, we can find lower bounds strictly greater than the diameter of the network topology [18, 19].

[3] The concept of line communication was later extended in various ways. For example, *wormhole routing* [7] is a hybrid of store-and-forward routing and circuit-switching routing. Several commercial parallel computer systems adopted the wormhole routing (see, e.g., [24] for a survey), and several papers address theoretical problems on it [10, 13, 21, 23].

[4] Hromkovič et al. considered a similar problem under a quite different model of circuit switching. Their model assumes that every node on a delivery route can eavesdrop on the message flowing on the route. See [15] for details.

same time complexity as the first one, provided that $1/(1-\epsilon) = \Omega(n)$. By combining these algorithms, we obtain an asymptotically optimal group-gossiping algorithm for any U ($\subseteq V$).

Thus far, several routing problems have been investigated under the circuit-switching routing model [1, 6, 16, 17], including the broadcast problem [5, 9]. To the authors' knowledge, however, no investigation has been carried out for the gossiping problem under the circuit-switching model.

This paper is organized as follows. In §2, we introduce some notation and define the circuit-switching model and the group-gossiping problem formally. We also show a lower bound on the time complexity of the group-gossiping problem. Sections 3, 4, and 5 are concerned with the first gossiping algorithm, which asymptotically achieves the lower bound when $1/(1-\epsilon) = o(n)$. In §3, we show an outline of the first algorithm. It consists of three phases. Sections 4 and 5 describe an efficient implementation of each phase. Section 6 proposes the second algorithm, which asymptotically achieves the lower bound when $1/(1-\epsilon) = \Omega(n)$. The algorithm uses the procedure proposed in §5; the procedure of §5 is commonly used in both algorithms. Section 7 concludes the paper.

2. Preliminaries.

2.1. Notation.

Let $\mathcal{H} = (V, E)$ be an undirected n-cube, where $V = \{0,1\}^n$, and for any $u, v \in V$, $\{u, v\} \in E$ iff u and v differ in exactly one bit. An element in V is called a *node*, and an element in E is called an *edge*. If u and v differ in the ith bit[5] and $\{u, v\} \in E$, we denote $u = \oplus_i v$. (Note that if $u = \oplus_i v$, then $v = \oplus_i u$.) An edge $\{u, v\}$ is called the ith edge of u if $v = \oplus_i u$. Let $E_i \subseteq E$ be the set of the ith edges of $\mathcal{H}$. An ordered pair (u, v) of nodes is called a *link*. If $\{u, v\} \in E_i$, then (u, v) is called the ith link of u. Let $L_i = \{(u, v), (v, u) : \{u, v\} \in E_i\}$ be the set of ith links. Finally, let $L = \bigcup_{1 \le i \le n} L_i$ be the set of all links.

DEFINITION 2.1. *Let m be an integer in $\{0, 1, \ldots, n\}$. For each $x \in \{0,1\}^m$, let $\mathcal{H}_x$ be the subcube of $\mathcal{H}$ induced by the set of nodes $\{yx : y \in \{0,1\}^{n-m}\}$.*

DEFINITION 2.2. *Let v be a node in V and m be an integer in $\{0, 1, \ldots, n\}$. For each $x \in \{0,1\}^m$, $v(x)$ is defined as $v(x) = yx$, where y is the prefix of v of length $n - m$. For given $U \subset V$, $U(x)$ is defined as $U(x) = \{v(x) : v \in U\}$.*

Example 2.3. Let $n = 4$. $\mathcal{H}_{01}$ is the 2-cube induced by set {0001,0101,1001,1101}. When $v = 1111$ and $x = 01$, $v(x) = 1101$ since $v(x)$ is the node which has the same prefix 11 of length 2 with v and suffix x.

DEFINITION 2.4. *Given $G = (V, E)$, a* path *connecting u and v ($\in V$) is a sequence of links $P = \ell_1 \ell_2 \ldots \ell_m$ such that $\ell_i = (w_{i-1}, w_i) \in L$ for $1 \le i \le m$, $w_0 = u$, and $w_m = v$. If the w_i's are distinct, path P is said to be* simple. *Since we consider only simple paths in this paper, a path means a simple path. If a path P contains a link (u, v) and another path Q contains a link (u, v) or (v, u), then they are said to* share *edge $\{u, v\}$. A set of paths are said to be* edge-disjoint *if any two paths in the set share no edges.*

Let U and W be two disjoint subsets of V. We say that U and W are connected by edge-disjoint paths *if there is a set of edge-disjoint paths such that each node in U (resp. in W) is connected with a node in W (resp. in U) by some path in the set. If there is a set of edge-disjoint paths which contains a path connecting u and v for*

[5] In this paper, we number bits in binary strings from left to right. That is, the most significant bit is the first bit and the least significant bit is the nth bit, where n is the length of the binary string.

any $u \in U$ and $v \in W$, then U and W are said to be fully connected by edge-disjoint paths. *When $\{u\}$ and U are connected by edge-disjoint paths, we say that u and U are connected by edge-disjoint paths.*

2.2. Circuit-switching model. In the rest of this paper, with a graph $G = (V, E)$ we identify a parallel computer system consisting of the set V of nodes connected by the set E of links. A node in $\mathcal{H}$ communicates with another node in $\mathcal{H}$ by sending a message along with a path connecting them in $\mathcal{H}$. $\mathcal{H}$ has a global clock, and all nodes synchronously execute their operations according to the global clock. More precisely, each node can initiate communication at any time instant $t = 0, 1, \ldots$. When a node u wishes to send a message to another node v at some time instant t_0, u first selects a path connecting u and v as the message route. Let $\mathcal{P}$ be the set of paths selected by nodes who wish to initiate communication at the same time instant t_0. If a path in $\mathcal{P}$ is edge-disjoint with any other paths in $\mathcal{P}$, then the message is sent from its source to its destination through the path by (and not including) time $t_0 + 1$. If two paths share an edge, one of them is selected arbitrarily, and communication using the selected path occurs. A node u can send out messages to *all* edges incident on u simultaneously. Moreover, those messages can be distinct. However, u never receives a message from an edge to which it sends out a message since paths must be edge-disjoint. The message sent is received only by the destination node at the other end of the path. Other intermediate nodes in the path cannot receive the message, namely, they merely relay the message.

Finally, we assume that each message is long enough to carry any number of tokens.[6]

A time interval $[t, t + 1)$ is called a *time unit*, and we assume that nodes can send and/or receive messages as described in the previous paragraph and prepare the next communication in one time unit.

We call this model the *circuit-switching model.* Note that the time required for reserving a path is assumed to be negligible in this model. It is equivalent to assuming that a time unit is long enough to complete both reservation and message transmission. Note also that under this model, edges are *half-duplex*, i.e., at most one message can pass through in one time unit.

2.3. Group-gossiping problem. Let $\mathcal{H} = (V, E)$ be an n-cube. For a given nonempty subset $U \subseteq V$, consider the following problem: broadcast $\{t(u) : u \in U\}$ to all nodes in U, where $t(u)$ is a piece of information called token held by $u \in U$. We call this problem the *group-gossiping problem* for U. When $|U| = 2$, it is the point-to-point routing problem, and when $U = V$, it is the conventional gossiping problem [14]. In what follows, let $|U| = 2^{\epsilon n}$ for $0 < \epsilon \leq 1$.

A lower bound on the group-gossiping time under the circuit-switching model is derived as follows. Under the model, a node $v \in V$ can send any number of tokens to at most n nodes in a time unit since the degree of v is n. (Note that some of the n receivers may not be adjacent with v.) Hence it requires at least

$$\left\lceil \frac{\log_2(|U| - 1)}{\log_2 n} \right\rceil$$

[6] Standard literature (see [14]) on the gossiping problem adopts this assumption. In practice, sharing raw large information is inefficient in both communication time and space. Hence, an important algorithm-design issue is who shares what. As a result, small "key" information is shared by nodes, as usual. The assumption reflects it.

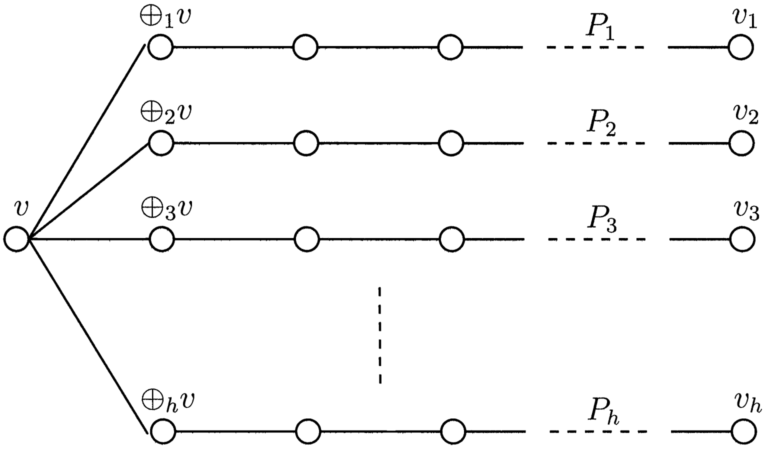


FIG. 1. *An illustration for the proof of Lemma* 3.1.

time units to broadcast token $t(v)$ to all nodes in $U \setminus \{v\}$. Since group gossiping requires at least the same time units, we have the following theorem.

THEOREM 2.5 (lower bound). *The group gossiping for U requires at least $\lceil \log_2(|U|-1)/\log_2 n \rceil$ time units.*

3. Outline of algorithm GROUP_GOSSIP1.

This section describes an outline of the first group-gossiping algorithm, GROUP_GOSSIP1, which asymptotically achieves the lower bound in Theorem 2.5, provided $1/(1-\epsilon) = o(n)$. In §§3–5, we assume that $1/(1-\epsilon) = o(n)$.

We show a lemma which prepares the basis for how to distribute and collect information from n other nodes in the n-cube under the circuit-switching model.

LEMMA 3.1. *Let h be any natural number and let $\mathcal{K} = (X, A)$ be an h-cube. Then any set $Y \subseteq X$ of size h and any node v $(\in X \setminus Y)$ are connected by edge-disjoint paths.*

Proof. Without loss of generality, we can assume $v = 0^h$. Let $Y = \{v_1, v_2, \ldots, v_h\}$ be any h-set which does not contain $v = 0^h$.

See Figure 1 for an illustration. For each $1 \le i \le h$, condsider the shortest path P_i connecting $\oplus_i v$ and v_i defined as follows: $P_i = \ell_1 \ell_2 \ldots \ell_m$, where $\ell_k \in L_{j_k}$ for $1 \le k \le m$, and $L_{j_1} L_{j_2} \ldots L_{j_m}$ is a subsequence of $L_{i+1} L_{i+2} \ldots L_h L_1 \ldots L_i$. Note that $B = b_1 b_2 \ldots b_m$ is a subsequence of $C = c_1 c_2 \ldots c_h$ iff B is obtained from C by removing some c_i's. In short, path P_i uses links in $L_{i+1} L_{i+2} \ldots L_h L_1 \ldots L_i$ in this order. Clearly, P_i is determined uniquely since for each $1 \le k \le h$, links in L_k occur at most once in P_i.

First, we show that any two paths P_i and P_j $(i \ne j)$ do not contain the same link in L. Suppose that both P_i and P_j contain a link $\ell = (w, \oplus_k w) \in L_k$ for some k. Without loss of generality, we assume that $i < j$. By definition, $\oplus_i v$ and $\oplus_j v$ differ at the ith bit. Since a link in L_i occurs as the last link in P_i (if it occurs), the ith bits of w and $\oplus_i v$ are the same. Therefore, in P_j, a link in L_i must occur before $\ell \in L_k$. However, this is impossible since $i < j$. Hence if P_i and P_j share an edge $\{u, v\}$, one contains link (u, v) and the other contains link (v, u).

We construct a set of edge-disjoint paths $\mathcal{P}$ connecting the sets $\{\oplus_i v : 1 \le i \le h\}$ and $Y = \{v_i : 1 \le i \le h\}$ by modifying P_i's as follows: (1) Select an edge $\{w, \oplus_k w\}$

shared by two paths P_i and P_j. Let $P_i = Q_1(w, \oplus_k w)Q_2$ and $P_j = Q_3(\oplus_k w, w)Q_4$, where Q_1, Q_2, Q_3, and Q_4 are sequences of links. (2) Modify these paths as $P_i = Q_1Q_4$ and $P_j = Q_3Q_2$, i.e., remove the shared edge by exchanging their destinations. (3) If there remains a shared edge, go to (1); otherwise, let $\mathcal{P}$ be the set of P_i's and stop.

In step (2), a shared edge is removed from the set of P_i's. Hence the procedure terminates. It is clear that when it terminates, paths in $\mathcal{P}$ are edge-disjoint. Now we have a set of edge-disjoint paths connecting $\{\oplus_1 v, \oplus_2 v, \ldots, \oplus_h v\}$ and Y. For any i, P_i contains no links incident on v because $v \notin Y$. Hence, by catenating link $(v, \oplus_i v)$ to P_i for each i, we have edge-disjoint paths connecting v and Y. □

Algorithm GROUP_GOSSIP1 consists of three phases. Each node u in U initially holds a distinct token $t(u)$. The first phase moves these $|U|$ tokens to a set W of $2^d (\leq |U|)$ nodes, called *intermediate nodes*, in one time unit. Here d is an integer determined by $|U|$ and n. This move uses the communication routes constructed in Lemma 3.1. At the end of the first phase, every intermediate node ($\in W$) keeps at least $\lfloor |U|/2^d \rfloor$ tokens. In the second phase, the nodes in W exchange the tokens they collected in the first phase by repeatedly applying a communication scheme proposed in §5. As we will see later, it takes $\log_2 |U| / \log_2 n + o(\log_2 |U| / \log_2 n)$ time units, and after the second phase, every node in W holds the set $T = \{t(u) : u \in U\}$ of all tokens. Finally, in the third phase, the nodes in W broadcast T to all nodes in U in one time unit by using the communication routes in the first phase in the reverse direction.

Algorithm GROUP_GOSSIP1 is described as follows.

ALGORITHM GROUP_GOSSIP1 (for U satisfying $1/(1-\epsilon) = o(n)$)

Phase 1. Each node $u \in U$ sends $t(u)$ to a node in W in one time unit.

Phase 2. All nodes in W exchange their tokens in $\log_2 |U| / \log_2 n + o(\log_2 |U| / \log_2 n)$ time units.

Phase 3. Every node in U receives the set of all tokens from a node in W in one time unit.

The following two sections describe concrete implementations of these three phases. In §4, we determine the set W of intermediate nodes for given U and construct edge-disjoint paths connecting U and W. Section 5 shows an algorithm for exchanging tokens among all nodes in W quickly.

4. Communication with intermediate nodes.

In this section, we first determine the set of intermediate nodes W and then construct edge-disjoint paths connecting U and W. The edge-disjoint paths are used as the communication routes in both Phases 1 and 3: tokens flow from U to W in Phase 1 and from W to U in Phase 3 through the same routes.

4.1. Intermediate nodes W.

Given U ($\subset V$), let d be the smallest nonnegative integer which satisfies the inequality

$$\left\lceil \frac{|U|}{2^d} \right\rceil \leq n - d. \tag{1}$$

Inequality (1) does not hold when $d = n$ since U is assumed to be nonempty. On the other hand, inequality (1) holds when $d = n - 1$ since $1/(1-\epsilon) < n$. Then d is well-defined and $d < n$.

Consider the following procedure, PARTITION, and let $\mathcal{U}$ be the partition of U generated by PARTITION(n, U). Since PARTITION includes nondeterministic choice, $\mathcal{U}$ may not be determined uniquely. In such a case, we select an arbitrary one as $\mathcal{U}$ and fix it.

```
PROCEDURE PARTITION($m, U_x$)
    Step 1. If $m \leq n - d$, then return $U_x$.
    Step 2. Partition $U_x$ into the following three subsets:
```

$$\begin{aligned} X_0 &= \{w0y \in U_x : w1z \in U_x, |w| = m-1 \text{ and } y, z \in \{0,1\}^{n-m}\}, \\ X_1 &= \{w1y \in U_x : w0z \in U_x, |w| = m-1 \text{ and } y, z \in \{0,1\}^{n-m}\}, \quad \text{and} \\ Y &= U_x \setminus (X_0 \cup X_1). \end{aligned}$$

Note that for any element in X_0, there is an element having the same prefix of length $m-1$ in X_1, and vice versa.

```
    Step 3. Let $Y_0$ be an arbitrary subset of $Y$ of size $\lceil |Y|/2 \rceil$. Let $Y_1 = Y \setminus Y_0$.
    Step 4. Let $U_{0x} = X_0 \cup Y_0$ and $U_{1x} = X_1 \cup Y_1$.
    Step 5. Call PARTITION($m-1, U_{0x}$) and PARTITION($m-1, U_{1x}$).
```

Example 4.1. Consider the case of $n = 4$. Let

$$U = \{1110, 1010, 0001, 0101, 1001, 1111\}.$$

Let us partition U by calling PARTITION$(4, U)$ and obtain $\mathcal{U}$. The smallest integer d satisfying inequality (1) is 1. Since $4\ (= m)\ > 3\ (= n - d)$, it proceeds to Step 2. In Step 2, U is partitioned into three subsets $X_0 = \{1110\}$, $X_1 = \{1111\}$, and $Y = \{1010, 0001, 0101, 1001\}$. In Step 3, Y is further partitioned into two subsets $Y_0 = \{1010, 0001\}$ and $Y_1 = \{0101, 1001\}$, for example, and then in Step 4, U_0 and U_1 are determined as

$$\begin{aligned} U_0 &= X_0 \cup Y_0 = \{1110, 1010, 0001\} \quad \text{and} \\ U_1 &= X_1 \cup Y_1 = \{1111, 0101, 1001\}. \end{aligned}$$

In Step 5, PARTITION$(3, U_0)$ and PARTITION$(3, U_1)$ are called. These procedures return U_0 and U_1 since $3\ (= m) = n - d$.

The following lemma shows that PARTITION partitions a given set U into subsets with almost equal sizes.

LEMMA 4.2. *For any two subsets $U_x, U_y \in \mathcal{U}$, $|U_x|$ and $|U_y|$ differ at most* 1.

Proof. PARTITION partitions U by recursively calling itself. The proof is by induction on the level t of the recursion. When $t = 1$, since $|X_0| = |X_1|$, $||U_0| - |U_1|| \leq 1$ holds by Step 3. Assume that for any two subsets U_x and U_y constructed at level t, $||U_x| - |U_y|| \leq 1$ holds. If each subset constructed at level t has an equal size, since each U_x is partitioned into two subsets U_{0x} and U_{1x} such that $||U_{0x}| - |U_{1x}|| \leq 1$ in Steps 3 and 4 at level $t + 1$, the statement holds. Suppose that $|U_x| = k$ and $|U_y| = k + 1$ for some U_x and U_y. If k is even, since $|U_{0x}| = |U_{1x}| = k/2$ and $\max\{|U_{0y}|, |U_{1y}|\} = k/2 + 1$, the statement holds. If k is odd, since $|U_{0y}| = |U_{1y}| = (k+1)/2$ and $\min\{|U_{0x}|, |U_{1x}|\} = (k-1)/2$, the statement again holds. Hence the lemma holds. □

By Lemma 4.2, we immediately obtain the following corollary, since the maximum level of recursion is d and hence $|\mathcal{U}| = 2^d$.

COROLLARY 4.3. *The size of any subset obtained by* PARTITION *is at most* $\lceil |U|/2^d \rceil$.

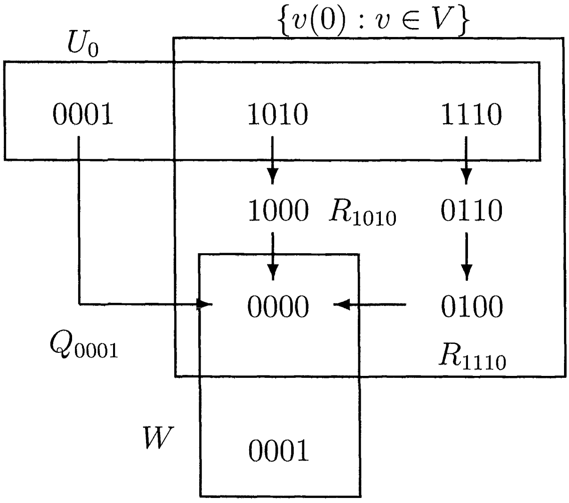


FIG. 2. *An illustration of edge-disjoint paths connecting U_0 and W.*

Using the partition $\mathcal{U}$, we define W as follows:

$$W = \{0^n(x) : U_x \in \mathcal{U}\}.$$

Recall the definition of notation $v(x)$ given in Definition 2.2.

Example 4.4. Consider Example 4.1 again. Since $\mathcal{U} = \{U_0, U_1\}$, $W = \{0000(0), 0000(1)\} = \{0000, 0001\}$.

4.2. Edge-disjoint paths connecting U and W. This subsection presents edge-disjoint paths connecting U and W.

Let v be an arbitrary node in U. Suppose that v is in $U_x \in \mathcal{U}$. Let $u = 0^n(x)$ be a node in W. We will connect v and u by a path $P_v = Q_v R_v$ which is determined as follows:

(i) Q_v is the shortest path connecting v and $v(x)$ in such a way that if in Q_v a link in L_x occurs before a link in L_y, then $x > y$ holds. In short, Q_v uses links in $L_n, L_{n-1}, \ldots, L_{n-|x|+1}$ in this order.

(ii) R_v connects $v(x)$ and u. Notice that both $v(x)$ and u $(= 0^n(x))$ are in $\mathcal{H}_x$. (Recall Definition 2.1 for $\mathcal{H}_x$.) Since $|\{v(x) : v \in U_x\}| \leq \lceil |U|/2^d \rceil \leq n - d$ by Corollary 4.3, there are edge-disjoint paths connecting u and each of $v(x)$'s by Lemma 3.1. We take the path as R_v.

Example 4.5. See Figure 2 for an illustration. Consider the set of intermediate nodes $W = \{0000, 0001\}$ obtained in Example 4.4. In this case, 0001 $(\in U_0)$ and 0000 $(= 0001(0))$ are connected by $Q_{0001} = (0001, 0000)$. Since $0000 \in U_0(0)$ is an element of W, $U_0(0) \setminus \{0000\}$ is connected with 0000 by paths

$$R_{1110} = (1110, 0110)(0110, 0100)(0100, 0000) \quad \text{and}$$
$$R_{1010} = (1010, 1000)(1000, 0000).$$

Note that in this particular case, since 1010 and 1110 are in $\mathcal{H}_0$, both Q_{1010} and Q_{1110} are empty paths, and since 0000 is in W, R_{0001} is also an empty path.

Let $\Pi_1 = \{Q_v : v \in U\}$, $\Pi_2 = \{R_v : v \in U\}$, and $\Pi = \{Q_v R_v : v \in U\}$. The goal of this subsection is to prove the following theorem.

THEOREM 4.6. *Any two paths $P_u, P_v \in \Pi$ are edge-disjoint.*

Theorem 4.6 is immediate, if all of the following three claims hold.

(i) Any two paths $Q_u \in \Pi_1$ and $R_v \in \Pi_2$ are edge-disjoint.

(ii) Any two paths $Q_u, Q_v \in \Pi_1$ are edge-disjoint.

(iii) Any two paths $R_u, R_v \in \Pi_2$ are edge-disjoint.

In the following we show the correctness of the three claims.

LEMMA 4.7. *Any two paths $Q_u \in \Pi_1$ and $R_v \in \Pi_2$ are edge-disjoint.*

Proof. Let U_x be any set in $\mathcal{U}$, and consider any node v in U_x. Note that $x \in \{0,1\}^d$. Since v and $v(x)$ have the same prefix of length $n-d$ and Q_v is a shortest path connecting them, Q_v does not use links in $\bigcup_{1\le i\le n-d} L_i$. On the other hand, since path R_v consists of links in $\mathcal{H}_x$, R_v does not use links in $\bigcup_{n-d<i\le n} L_i$. Hence for any $v, u \in U$, Q_v and R_u do not share an edge. □

LEMMA 4.8. *Any two paths $R_u, R_v \in \Pi_2$ are edge-disjoint.*

Proof. If u and v belong to the same $U_x \in \mathcal{U}$, R_u and R_v are edge-disjoint by definition. Suppose that u and v belong to different sets U_x and $U_y \in \mathcal{U}$, respectively. Since R_u and R_v are paths in different hypercubes $\mathcal{H}_x$ and $\mathcal{H}_y$, respectively, they do not share an edge. □

LEMMA 4.9. *Any two paths $Q_u, Q_v \in \Pi_1$ are edge-disjoint.*

Proof. Let $u \in U_x \in \mathcal{U}$ and $v \in U_y \in \mathcal{U}$ for some x and y (x and y may not be distinct). Suppose that Q_u and Q_v share an edge $e = \{w, \oplus_k w\} \in E_k$. Let us rewrite w as $w_1 b w_2$, where $|w_1| = k-1$, $b \in \{0,1\}$, and $|w_2| = n-k$. Then, using w_1, we rewrite u and v as follows:

$$u = w_1 b_1 z_1 \quad \text{and} \quad v = w_1 b_2 z_2,$$

where $b_1, b_2 \in \{0,1\}$, and $z_1, z_2 \in \{0,1\}^{n-k}$. Also using w_2, we rewrite x and y as follows:

$$x = x_1 w_2 \quad \text{and} \quad y = y_1 w_2,$$

where $x_1, y_1 \in \{0,1\}^{d+k-n}$.

First, consider the case of $b_1 \ne b_2$. By the definition of PARTITION, since x and y have suffix w_2 in common, both u and v belong to U_{w_2}. Therefore, $u = w_1 b_1 z_1 \in U_{b_1 w_2}$ and $v = w_1 b_2 z_2 \in U_{b_2 w_2}$ since $b_1 \ne b_2$. Since $b_1 w_2$ is a suffix of x, the kth bits of u and $u(x)$ are the same. Hence a link $\ell \in L_k$ never occurs in Q_u—a contradiction.

Next, consider the case of $b_1 = b_2$. Let w_3 be the longest common prefix of u and v. That is,

$$u = w_3 b_3 z_3 \quad \text{and} \quad v = w_3 b_4 z_4,$$

where $|w_3| \ge k$, $b_3, b_4 \in \{0,1\}$, $b_3 \ne b_4$, $z_3, z_4 \in \{0,1\}^h$, and $h < n-k$. Let w_4 be the suffix of w_2 with length h. Since both u and v belong to U_{w_2}, they both belong to U_{w_4}, as well. By the definition of PARTITION$(n-h, U_{w_4})$,

$$u \in U_{b_3 w_4} \quad \text{and} \quad v \in U_{b_4 w_4}$$

—a contradiction since either u or v is not in U_{w_2}. □

By using the edge-disjoint paths Π connecting U and W, Phases 1 and 3 are executed as follows.

Phase 1. Each $u \in U_x \in \mathcal{U}$ sends $t(u)$ to $w = 0^n(x) \in W$ through path P_u in Π connecting u and w.

Phase 3. Each $u \in U_x \in \mathcal{U}$ receives $T = \{t(v) : v \in U\}$ from $w = 0^n(x) \in W$ through path P_u in reverse direction.

Since Π is a set of edge-disjoint paths, we have the following theorem.

THEOREM 4.10. *Each of Phases* 1 *and* 3 *requires only one time unit.*

5. Exchange tokens among intermediate nodes.

Recall that set W of intermediate nodes satisfies that $|W| = 2^d$ and $d \leq n-1$. At the beginning of Phase 2, for each $\mathcal{H}_x$, tokens of nodes in U_x are held by node $0^{n-d}x$ in $\mathcal{H}_x$. Let $r = n-d-1$, and $s = \lfloor \log_2 r \rfloor = \lfloor \log_2(n-d-1) \rfloor$. In this section, we assume that $d \leq n-3$ without loss of generality. If $d \geq n-2$, let $Y = \{0^3x :\in \{0,1\}^{n-3}\} \subseteq W$. Then we regard Y as W in the rest of the section at the expense of at most six time units since all nodes $yx \in W$ can send tokens to $0^3x \in Y$ in three time units, as shown in *Preprocessing* below, and 0^3x can broadcast T to all nodes yx in three time units, as shown in *Postprocessing* below.

Preprocessing (when $d \geq n-2$). For $i = 1,2,3$, one in a time unit sequentially, for all $u \in W$ in parallel, if the ith bit of u is 1, then u sends all tokens it holds to $\oplus_i u$.

Postprocessing (when $d \geq n-2$). For $i = 3,2,1$, one in a time unit sequentially, for all $u = 0^i z \in W$ (for some z) in parallel, u sends set T of all tokens to $\oplus_i u$.

5.1. Basic communication routes.

Fix any $w \in \{0,1\}^{d-s}$ and consider the following two subsets:

$$S_0 = \{00^r xw : x \in \{0,1\}^s\} \quad \text{and}$$
$$S_1 = \{10^r xw : x \in \{0,1\}^s\}.$$

For each pair (u,v) in $S_0 \times S_1$, we give a path P_{uv} connecting u and v in $\mathcal{H}_w$ and show that the set of paths $\Gamma = \{P_{uv} : u \in S_0 \text{ and } v \in S_1\}$ is a set of edge-disjoint paths. For each $x \in \{0,1\}^*$, $\bar{x}$ denotes the integer whose binary representation is x. By $0^{(r;i_1,i_2,\ldots,i_j)}$, we denote the bit sequence $b_1 b_2 \ldots b_r \in \{0,1\}^r$ such that $b_k = 0$ iff $k \neq i_1, i_2, \ldots, i_j$.

Let $u = 00^r yw \in S_0$ and $v = 10^r zw \in S_1$. If $s = 1$, u and v are connected by P_{uv} as follows. Note that $r = 2$ or 3 since $s = \lfloor \log_2 r \rfloor$.

1. If $\bar{z} = \bar{y}$, then u and v are adjacent with each other. We take

$$P_{uv} = (u,v).$$

2. If $\bar{y} = 0$ and $\bar{z} = 1$,

$$P_{uv} = (00^r 0w, 00^{(r;1)}0w)(00^{(r;1)}0w, 10^{(r;1)}0w)$$
$$(10^{(r;1)}0w, 10^{(r;1)}1w)(10^{(r;1)}1w, 10^r 1w).$$

3. If $\bar{y} = 1$ and $\bar{z} = 0$,

$$P_{uv} = (00^r 1w, 00^{(r;2)}1w)(00^{(r;2)}1w, 10^{(r;2)}1w)$$
$$(10^{(r;2)}1w, 10^{(r;2)}0w)(10^{(r;2)}0w, 10^r 0w).$$

It is easy to verify that S_0 and S_1 are fully connected by $\Gamma = \{P_{uv} : u \in S_0$ and $v \in S_1\}$ and that Γ is a set of edge-disjoint paths.

In the following, we consider the case of $s \geq 2$. When $s \geq 2$, the path P connecting u and v is determined as follows:

1. If $\bar{z} = \bar{y}$, we take

$$P_{uv} = (u, v).$$

2. If $\bar{y} > \bar{z}$, the path P_{uv} consists of four subpaths P_1, P_2, P_3, and P_4. The first part P_1 of P_{uv} is given as follows:

$$P_1 = (00^r yw, 00^{(r;\bar{z}+1)}yw)(00^{(r;\bar{z}+1)}yw, 00^{(r;\bar{z}+1,\bar{y}+1)}yw).$$

Note that $\bar{z}+1 < \bar{y}+1 \leq r$ since $s = \lfloor \log_2 r \rfloor$. The second part P_2 of P_{uv} consists of a link in L_1:

$$P_2 = (00^{(r;\bar{z}+1,\bar{y}+1)}yw, 10^{(r;\bar{z}+1,\bar{y}+1)}yw).$$

The third part P_3 of P_{uv} connects nodes $10^{(r;\bar{z}+1,\bar{y}+1)}yw$ and $10^{(r;\bar{z}+1,\bar{y}+1)}zw$ by the shortest path which uses links in $L_{r+2}, L_{r+3}, \ldots, L_{r+s+1}$ in this order. The last part P_4 of P_{uv} is given as follows:

$$P_4 = (10^{(r;\bar{z}+1,\bar{y}+1)}zw, 10^{(r;\bar{y}+1)}zw)(10^{(r;\bar{y}+1)}zw, 10^r zw).$$

3. If $\bar{y} < \bar{z}$, the first, second, and fourth parts P_1, P_2, and P_4 of P_{uv} are given in the same way as in the case of $\bar{y} > \bar{z}$. The third part P_3 of P_{uv} connects $u_1 = 10^{(r;\bar{z}+1,\bar{y}+1)}yw$ and $u_2 = 10^{(r;\bar{z}+1,\bar{y}+1)}zw$ as follows.

(a) If for each $1 \leq i \leq s$, the ith bits of y and z differ, i.e., the Hamming distance between u_1 and u_2 is s $(= |y|)$, then P_3 is given as the shortest path connecting u_1 and u_2 using links in $L_{r+2}, L_{r+3}, \ldots, L_{r+s+1}$ in this order, i.e., P_3 is the same as in the case of $\bar{y} > \bar{z}$.

(b) Otherwise, let k be an integer in $\{r+2, r+3, \ldots, r+s+1\}$ such that the kth bits of u_1 and u_2 take the same value. Then P_3 is given as

$$P_3 = (u_1, \oplus_k u_1)P_3'(\oplus_k u_2, u_2),$$

, where P_3' is a shortest path connecting $\oplus_k u_1$ and $\oplus_k u_2$.

Example 5.1. Let $n = 10$. Suppose that $d = 4$. Then U is partitioned into 8 $(= 2^3)$ subsets, and $s = \lfloor \log_2 7 \rfloor = 2$. By selecting 00 as w $(\in \{0,1\}^{d-s} = \{0,1\}^2)$, for example, S_0 and S_1 are as follows:

$$S_0 = \{000000x00 : x \in \{0,1\}^2\} \quad \text{and}$$
$$S_1 = \{100000x00 : x \in \{0,1\}^2\}.$$

First, consider the path P connecting $00^5 01w$ $(\in S_0)$ and $10^5 10w$ $(\in S_1)$. Since $\overline{01} < \overline{10}$, we use the third rule. $P = P_1P_2P_3P_4$ is given as follows:

$$\begin{aligned}
P_1 &= (0000000100, 0001000100)(0001000100, 0011000100),\\
P_2 &= (0011000100, 1011000100),\\
P_3 &= (1011000100, 1011001100)(1011001100, 1011001000), \quad \text{and}\\
P_4 &= (1011001000, 1010001000)(1010001000, 1000001000).
\end{aligned}$$

Path P passes through the subcube induced by the nodes which contain 1's at the third and fourth bits. On the other hand, the path $Q = Q_1Q_2Q_3Q_4$ connecting

$00^5 10w$ and $10^5 01w$ is given by the second rule since $\overline{10} > \overline{01}$.

$$\begin{aligned}
Q_1 &= (0000001000, 0010001000)(0010001000, 0011001000),\\
Q_2 &= (0011001000, 1011001000),\\
Q_3 &= (1011001000, 1011000000)(1011000000, 1011000100), \quad \text{and}\\
Q_4 &= (1011000100, 1001000100)(1001000100, 1000000100).
\end{aligned}$$

Path Q also passes through the subcube induced by the nodes which contain 1's at the third and fourth bits. However, P and Q do not share edges in the subcube since P_3 and Q_3 are selected to be edge-disjoint.

Let $\Gamma = \{P_{uv} : u \in S_0 \text{ and } v \in S_1\}$. It is obvious that each path P_{uv} in Γ correctly connects $u \in S_0$ and $v \in S_1$ using edges in $\mathcal{H}_w$. The following three lemmas guarantee that Γ is a set of edge-disjoint paths.

LEMMA 5.2. *No edge in E_1 is shared by paths in Γ.*

Proof. Let $P \in \Gamma$ be the path connecting a node $00^r yw \in S_0$ and a node $10^r zw \in S_1$.

First, consider the case of $y \neq z$. Assume that there exists a path Q ($\neq P$) in Γ which shares an edge $\{\alpha, \oplus_1 \alpha\}$ in E_1 with P. Without loss of generality, we assume that P contains link $\ell = (\alpha, \oplus_1 \alpha)$. Then Q must contain the same link $(\alpha, \oplus_1 \alpha)$ since each path in Γ contains exactly one link in L_1 and every node in S_0 has the same prefix 0. (In other words, all edges in E_1 are used in the same direction.) Since α has prefix $00^{(r;\bar{z}+1,\bar{y}+1)}$, Q must connect $00^r zw$ and $10^r yw$. Hence $00^r yw$ and $00^r zw$ have the same suffix of length d with α—a contradiction since $y = z$ follows.

Next, consider the case of $y = z$. Let $Q \in \Gamma$ be any path ($\neq P$) which shares an edge in E_1 with P. Since P contains link $(00^r yw, 10^r yw)$ in L_1, Q also connects $00^r yw$ and $10^r yw$—a contradiction. Hence the lemma holds. □

LEMMA 5.3. *Let $\mathcal{E}_1 = \bigcup_{2 \le i \le r+1} E_i$. No edge in $\mathcal{E}_1$ is shared by paths in Γ.*

Proof. Let $P \in \Gamma$ be the path connecting a node $00^r yw \in S_0$ and a node $10^r zw \in S_1$. In each path in Γ, edges in $\mathcal{E}_1$ are used in the first and fourth parts. Since every node in S_0 has the same prefix 0 and each path in Γ uses exactly one link in L_1, for any two paths $P, Q \in \Gamma$, the first part of P and the fourth part of Q are edge-disjoint.

We prove that for any two paths $P, Q \in \Gamma$, the first parts of P and Q are edge-disjoint. (We can apply a similar argument to the fourth part, so we omit the proof for that case.)

The first part of P contains two edges in $\mathcal{E}_1$, $\{00^r yw, 00^{(r;\bar{z}+1)} yw\}$ and $\{00^{(r;\bar{z}+1)} yw, 00^{(r;\bar{z}+1,\bar{y}+1)} yw\}$. Therefore, if Q contains one of the above edges, then Q must connect $00^r yw$ and $00^r zw$—a contradiction. Hence the lemma holds. □

LEMMA 5.4. *Let $\mathcal{E}_2 = \bigcup_{r+2 \le i \le r+s+1} E_i$. No edge in $\mathcal{E}_2$ is shared by paths in Γ.*

Proof. Let $P \in \Gamma$ be the path connecting a node $00^r yw \in S_0$ and a node $10^r zw \in S_1$. Since if $y = z$, P uses no edges in $\mathcal{E}_2$, without loss of generality, we assume $y \neq z$.

Assume that there exists a path Q ($\neq P$) in Γ which shares an edge e in $\mathcal{E}_2$ with P. Since $y \neq z$, by definition, e connects two nodes having the same prefix $10^{(r;\bar{z}+1,\bar{y}+1)}$. Hence Q connects $00^r zw$ and $10^r yw$. Without loss of generality, we assume $\bar{y} > \bar{z}$. Let

$$u_1 = 10^{(r;\bar{z}+1,\bar{y}+1)} yw \quad \text{and} \quad u_2 = 10^{(r;\bar{z}+1,\bar{y}+1)} zw.$$

Since $\bar{y} > \bar{z}$, the third part P_3 of P connects u_1 and u_2 by the shortest path using links in $L_{r+2}, L_{r+3}, \ldots, L_{r+s+1}$ in this order.

If the Hamming distance between y and z is s (≥ 2), the third part Q_3 of Q connects u_2 and u_1 by the shortest path which uses links in $L_{r+2}, L_{r+3}, \ldots, L_{r+s+1}$ in this order. Let $e = \{10^{(r;\bar{z}+1,\bar{y}+1)}\alpha w, \oplus_{r+k+1}10^{(r;\bar{z}+1,\bar{y}+1)}\alpha w\}$. Let us rewrite α as $\alpha_1 b \alpha_2$, where $|\alpha_1| = k-1$, $|\alpha_2| = s-k$, and $b \in \{0,1\}$. In P, α_1 is a prefix of z and α_2 is a suffix of y. On the other hand, in Q, α_1 is a prefix of y and α_2 is a suffix of z. Therefore, for some $b_1, b_2 \in \{0,1\}$, $y = \alpha_1 b_1 \alpha_2$ and $z = \alpha_1 b_2 \alpha_2$—a contradiction since the Hamming distance between y and z is at most 1.

If the Hamming distance between y and z is strictly less than s, the third part Q_3 of Q is given as follows:

$$Q_3 = (u_2, \oplus_{r+k+1}u_2)Q'_3(\oplus_{r+k+1}u_1, u_1),$$

where k is an integer such that y and z take the same value at the kth bit. Let $b \in \{0,1\}$ be the kth bit of y. Since P_3 is the shortest path connecting u_1 and u_2, for any edge $\{\alpha_1, \alpha_2\}$ used in P_3, the $(r+k+1)$st bits of α_1 and α_2 are b. On the other hand, let $\{\beta_1, \beta_2\}$ be an edge used in Q_3. Then either the $(r+k+1)$st bit of β_1 or that of β_2 is not b since Q'_3 is a shortest path connecting $\oplus_{r+k+1}u_2$ and $\oplus_{r+k+1}u_1$. Hence P_3 and Q_3 are edge-disjoint. □

By Lemmas 5.2, 5.3, and 5.4, we have the next theorem.

THEOREM 5.5. *Sets S_0 and S_1 are fully connected by set Γ of edge-disjoint paths in $\mathcal{H}_w$.*

5.2. Exchange tokens among all nodes in W. Let σ be an integer in $\{0, 1, \ldots d-1\}$. For each $w_1 \in \{0,1\}^\sigma$ and $w_2 \in \{0,1\}^{d-(\sigma+s)}$, define three subsets W_{w_1,w_2}, $\overline{W_{w_1,w_2}}$, and $\overline{W}$ as follows:

$$\begin{aligned} W_{w_1,w_2} &= \{00^r w_1 x w_2 : x \in \{0,1\}^s\}, \\ \overline{W_{w_1,w_2}} &= \{10^r w_1 x w_2 : x \in \{0,1\}^s\}, \quad \text{and} \\ \overline{W} &= \{10^r z : 00^r z \in W\}. \end{aligned}$$

Then $\mathcal{W} = \{W_{w_1,w_2} : w_1 \in \{0,1\}^\sigma \text{ and } w_2 \in \{0,1\}^{d-(\sigma+s)}\}$ (resp. $\overline{\mathcal{W}} = \{\overline{W_{w_1,w_2}} : w_1 \in \{0,1\}^\sigma \text{ and } w_2 \in \{0,1\}^{d-(\sigma+s)}\}$) forms a partition of W (resp. $\overline{W}$) and $|\mathcal{W}| = |\overline{\mathcal{W}}| = 2^{d-s}$. By Theorem 5.5, W_{w_1,w_2} and $\overline{W_{w_1,w_2}}$ are fully connected by edge-disjoint paths in the subcube induced by $\{y w_1 x w_2 : y \in \{0,1\}^{r+1} \text{ and } x \in \{0,1\}^s\}$ because of the symmetry with respect to dimension. Let $\Gamma_{w_1,w_2} = \{Q_{uv} : u \in W_{w_1,w_2} \text{ and } v \in \overline{W_{w_1,w_2}}\}$ be the set of edge-disjoint paths guaranteed by Theorem 5.5. By using Γ_{w_1,w_2}'s, Phase 2 is realized as procedure TOKEN_EXCHANGE as follows.

PROCEDURE TOKEN_EXCHANGE

Step 1. Let $\sigma = 0$ and $b = 0$.

Step 2. Repeat Steps 3 and 4 $\lceil d/s \rceil$ times.

Step 3. If $b = 0$ (resp. $b = 1$), for all $w_1 \in \{0,1\}^\sigma$ and $w_2 \in \{0,1\}^{d-(\sigma+s)}$, each $u \in W_{w_1,w_2} \subseteq W$ (resp. $u \in \overline{W_{w_1,w_2}} \subseteq \overline{W}$) sends all tokens it holds to every node in $\overline{W_{w_1,w_2}}$ (resp. W_{w_1,w_2}) through paths in Γ_{w_1,w_2}.

Step 4. Let $\sigma = \sigma + s$ and $b = b \oplus 1$.

Step 5. If $\lceil d/s \rceil$ is odd, each node u in $\overline{W}$ sends the set of tokens it holds (which is shown to be $T = \{t(v) : v \in U\}$ in below) to node $\oplus_1 u$ in W by link $(u, \oplus_1 u) \in L_1$.

The key observation regarding TOKEN_EXCHANGE is that if $\lceil d/s \rceil$ is even, every node in W holds set $T = \{t(v) : v \in U\}$ of all tokens, and otherwise, if $\lceil d/s \rceil$ is odd, every node in $\overline{W}$ holds T when the loop of Steps 3 and 4 finishes since as the result of (a single execution of) Step 3, if $b = 0$ (resp. $b = 1$), then all tokens held in W_{w_1,w_2} (resp. $\overline{W_{w_1,w_2}}$) are sent to every node in $\overline{W_{w_1,w_2}}$ (resp. W_{w_1,w_2}). This is because W_{w_1,w_2} and $\overline{W_{w_1,w_2}}$ are fully connected by edge-disjoint paths (for a more formal discussion, see the proof of the following theorem).

THEOREM 5.6. TOKEN_EXCHANGE *correctly solves the group-gossiping problem for W in* $\log_2 |U| / \log_2 n + o(\log_2 |U| / \log_2 n)$ *time units.*

Proof. Let $u = 00^r x$ and $v = 00^r y$ be any two nodes in W. In order to show correctness, it is sufficient to show that there is a sequence of paths $P_1, P_2, \ldots, P_{\lceil d/s \rceil + 1}$ connecting u and v, where P_i is a path used in the ith round of TOKEN_EXCHANGE. (Since u and v are taken arbitrarily, this argument also shows the existence of a sequence of paths connecting v and u.)

Let $K = \lceil d/s \rceil$. We rewrite x and y as $x = x_1 x_2 \ldots x_K$ and $y = y_1 y_2 \ldots y_K$, where $|x_i| = |y_i| = s$ for $1 \le i \le K-1$ and $|x_K| = |y_K| \le s$. Since $W_{\lambda, x_2 \ldots x_K}$ and $\overline{W_{\lambda, x_2 \ldots x_K}}$ are fully connected by $\Gamma_{\lambda, x_2 \ldots x_K}$, there is a path P_1 in $\Gamma_{\lambda, x_2 \ldots x_K}$ which connects u with $u_1 = 10^r y_1 x_2 \ldots x_K$, where λ is an empty sequence. Since $W_{y_1, x_3 \ldots x_K}$ and $\overline{W_{y_1, x_3 \ldots x_K}}$ are fully connected by $\Gamma_{y_1, x_3 \ldots x_K}$, there is a path P_2 in $\Gamma_{y_1, x_3 \ldots x_K}$ which connects u_1 with $u_2 = 00^r y_1 y_2 x_3 \ldots x_K$, and so on. By repeating similar arguments, there is a path P_K which connects u_{K-1} with $u_K = b0^r y_1 y_2 \ldots y_K$, where $b = 0$, i.e., $u_K = 00^r y = v$ if K is even and $b = 1$ if K is odd. In the latter case, the tokens T held in u_K are sent to v $(= 00^r y)$ in Step 5 through path $P_{K+1} = (u_K, \oplus_1 u_K)$.

As for the number of time units, since any two paths $P \in \Gamma_{w_1,w_2}$ and $Q \in \Gamma_{w_1',w_2'}$ are edge-disjoint, each round of TOKEN_EXCHANGE completes in one time unit. Let us estimate $\lceil d/s \rceil$. Since $\lceil |U|/2^d \rceil \le n - d$ holds for $d = \epsilon n - 1$ and ϵn (because $1/(1-\epsilon) = o(n)$), by the definition of d, $d \le \epsilon n$. By the definition of s,

$$\begin{aligned} s &= \lfloor \log_2 (n - d - 1) \rfloor \\ &\ge \log_2 n(1-\epsilon) - 2 \\ &= \log_2 n + \log_2 (1-\epsilon) - 2. \end{aligned}$$

Since $d \le \epsilon n = \log_2 |U|$, we have

$$\begin{aligned} \lceil d/s \rceil &\le \left\lceil \frac{\log_2 |U|}{\log_2 n + \log_2 (1-\epsilon) - 2} \right\rceil \\ &\le \frac{\log_2 |U|}{\log_2 n + \log_2 (1-\epsilon) - 2} + 1. \end{aligned}$$

Since $1/(1-\epsilon) = o(n)$, $\log_2 n + \log_2 (1-\epsilon) = \log_2 n - o(\log_2 n)$. Now we have

$$\lceil d/s \rceil \le \frac{\log_2 |U|}{\log_2 n} + o\left(\frac{\log_2 |U|}{\log_2 n} \right). \qquad \square$$

Each of Phases 1 and 3 takes one time unit by Theorem 4.10. If $d \ge n - 2$, we need at most six more time units to apply TOKEN_EXCHANGE as mentioned at the beginning of this section. Consequently, we have the following theorem.

THEOREM 5.7. *Assuming* $1/(1-\epsilon) = o(n)$, *algorithm* GROUP_GOSSIP1 *solves the group-gossiping problem for U in*

$$\frac{\log_2 |U|}{\log_2 n} + o\left(\frac{\log_2 |U|}{\log_2 n} \right)$$

time units, which is asymptotically optimal.

6. The second algorithm, GROUP_GOSSIP2.

This section proposes algorithm GROUP_GOSSIP2, which asymptotically achieves the lower bound in Theorem 2.5, provided $1/(1-\epsilon) = \Omega(n)$.

If $1/(1-\epsilon) = \Omega(n)$, then $\epsilon = 1 - O(1/n)$, and therefore, $|U| = 2^{\epsilon n} = 2^{n-O(1)}$.

Let δ be an integer in $\{0, 1, \dots, n-3\}$. We now give algorithm GROUP_GOSSIP2.

Algorithm GROUP_GOSSIP2

Phase 1. Let $W = \{0^\delta x : x \in \{0,1\}^{n-\delta}\}$. For each $x \in \{0,1\}^{n-\delta}$, fix a shortest-path tree $\mathcal{T}_x$ in which every node in $\mathcal{H}_x$ is connected with $0^n(x)$ by a shortest path. (The shortest paths in $\mathcal{T}_x$ need not be edge-disjoint.) In each $\mathcal{H}_x$ ($x \in \{0,1\}^{n-\delta}$) in parallel, node $0^n(x)$ collects all tokens in $\mathcal{H}_x$ in δ time units through edges in $\mathcal{T}_x$.

Phase 2. Apply TOKEN_EXCHANGE to W to exchange tokens among all nodes in W. Since $\delta \le n-3$, by Theorem 5.7, it correctly finishes in $\lceil (n-\delta)/\log_2(\delta-1)\rceil$ time units.

Phase 3. In each $\mathcal{H}_x$ ($x \in \{0,1\}^{n-\delta}$) in parallel, node $u = 0^n(x)$ in W broadcasts the set of all tokens to all nodes in $\mathcal{H}_x$ in δ time units through edges in $\mathcal{T}_x$ again.

The correctness of algorithm GROUP_GOSSIP2 is clear, and it finishes in

$$2\delta + \frac{n-\delta}{\log_2(\delta-1)} + 1$$

time units. Since $\log_2|U| = n - O(1)$, $n - \delta < \log_2|U|$ holds for $\delta = \omega(1)$. Since $|U| = \Theta(|V|)$, $\log_2|U|/\log_2 n = \Theta(n/\log_2 n)$. Hence, by selecting δ to satisfy $\delta = o(n/\log_2 n)$ and $\delta = \omega(n^\epsilon)$ for any constant $\epsilon < 1$ (e.g., $\delta = n/(\log_2 n)^2$ satisfies this condition), GROUP_GOSSIP2 asymptotically achieves the lower bound in Theorem 2.5. That is, the following theorem holds.

Theorem 6.1. *Assuming* $1/(1-\epsilon) = \Omega(n)$, *algorithm* GROUP_GOSSIP2 *correctly solves the group gossiping problem for U in*

$$\frac{\log_2|U|}{\log_2 n} + o\left(\frac{\log_2|U|}{\log_2 n}\right)$$

time units, which is asymptotically optimal.

7. Concluding remarks.

In this paper, we introduced the group-gossiping problem and considered the problem in n-cubes under the circuit-switching model. A lower bound on the gossiping time is $\lceil \log_2(|U|-1)/\log_2 n\rceil$ and is achieved asymptotically for any U ($\subseteq V$) by two algorithms, GROUP_GOSSIP1 and GROUP_GOSSIP2.

REFERENCES

[1] W. Aiello, T. Leighton, B. Maggs, and M. Newman, *Fast algorithms for bit-serial routing on a hypercube*, in Proc. 2nd Symposium on Parallel Algorithms and Architectures, Association for Computing Machinery, New York, 1990, pp. 55–64; Math. Systems Theory, 24 (1991), pp. 253–271.

[2] A. Bagchi, S. L. Hakimi, J. Mitchem, and E. Schmeichel, *Parallel algorithms for gossiping by mail*, Inform. Process. Lett., 34 (1990), pp. 197–202.

[3] A. Bagchi, S. L. Hakimi, and E. Schmeichel, *Gossiping in a distributed network*, IEEE Trans. Comput., 42 (1993), pp. 253–256.

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

[5] S. Bitan and S. Zaks, *Optimal linear broadcast*, J. Algorithms, 14 (1993), pp. 288–315.

[6] G.-I. Chen and T.-H. Lai, *Constructing parallel paths between two subcubes*, IEEE Trans. Comput., 41 (1992), pp. 118–123.

[7] W. J. Dally and C. L. Seitz, *Deadlock-free message routing in multiprocessor interconnection network*, IEEE Trans. Comput., 36 (1987), pp. 547–553.

[8] R. C. Entringer and P. J. Slater, *Gossips and telegraphs*, J. Franklin Inst., 307 (1979), pp. 353–360.

[9] A. M. Farley, *Minimum-time line broadcast networks*, Networks, 10 (1980), pp. 59–70.

[10] S. Felperin, P. Raghavan, and E. Upfal, *A theory of wormhole routing in parallel computers*, in Proc. 33rd Foundations of Computer Science, IEEE Press, Piscataway, NJ, 1992, pp. 563–572.

[11] P. Fraigniaud, *Asymptotically optimal broadcasting and gossiping in faulty hypercube multiprocessor*, IEEE Trans. Comput., 41 (1992), pp. 1410–1419.

[12] P. Fraigniaud and E. Lazard, *Methods and problems of communication in usual networks*, Tech. report 91–33, Laboratoire d'Informatique du Parallélisme, École Normale Supérieure de Lyon, Lyon, France, 1991; Discrete Appl. Math., 53 (1994), pp. 79–133.

[13] C. J. Glass and L. M. Ni, *Adaptive routing in mesh-connected networks*, in Proc. 14th International Conference on Distributed Computer Systems, IEEE Press, Piscataway, NJ, 1992, pp. 12–19.

[14] S. M. Hedetniemi, S. T. Hedetniemi, and A. L. Liestman, *A survey of gossiping and broadcasting in communication networks*, Networks, 18 (1988), pp. 319–349.

[15] J. Hromkovič, R. Klasing, E. A. Stöhr, and H. Wagener, *Gossiping in vertex-disjoint paths mode in d-dimensional grids and planar graphs*, in Proc. 1st Annual European Symposium on Algorithms, T. Lengauer, ed., IEEE Press, Piscataway, NJ, 1993, pp. 200–211.

[16] T. Knight, *Technologies for low latency interconnection switches*, in Proc. 1st Symposium on Parallel Algorithms and Architectures, Association for Computing Machinery, New York, 1989, pp. 351–358.

[17] R. Koch, *Increasing the size of a network by a constant factor can increase performance by more than a constant factor*, in Proc. 29th Foundations of Computer Science, IEEE Press, Piscataway, NJ, 1988, pp. 221–230.

[18] D. W. Krumme, *Fast gossiping for the hypercube*, SIAM J. Comput., 21 (1992), pp. 365–380.

[19] D. W. Krumme, G. Cybenko, and K. N. Venkataraman, *Gossiping in minimal time*, SIAM J. Comput., 21 (1992), pp. 111–139.

[20] F. T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*, Morgan Kaufmann, San Francisco, 1992.

[21] D. H. Linder and J. C. Harden, *An adaptive and fault tolerant wormhole routing strategy for k-ary n-cubes*, IEEE Trans. Comput., 40 (1991), pp. 2–12.

[22] D. Richards and A. L. Liestman, *Generalization of broadcasting and gossiping*, Networks, 18 (1988), pp. 125–138.

[23] L. Schwiebert and D. N. Jayasimha, *Optimal fully adaptive wormhole routing for meshes*, Tech. report OSU-CISRC-4/93-TR16, Department of Computer Science, Ohio State University, Columbus, OH, 1993.

[24] A. Trew and G. Wilson, eds., *Past, Present, Parallel: A Survey of Available Parallel Computer Systems*, Springer-Verlag, Berlin, New York, Heidelberg, 1991, pp. 125–147.

SIAM J. COMPUT.
Vol. 25, No. 5, pp. 1061–1081, October 1996

007

ON POINT LOCATION AND MOTION PLANNING AMONG SIMPLICES*

MARCO PELLEGRINI[†]

Abstract. Let U be a set of n possibly intersecting $(d-1)$-simplices in d-space for $d \geq 2$, and let $\mathcal{A}(U)$ be the arrangement of U. Let $K = |\mathcal{A}(U)|$ be the number of faces of any dimension in the arrangement of U. A data structure is described that uses storage $O(n^{d-1+\epsilon} + K)$ and is built *deterministically* in time $O(n^{d-1+\epsilon} + K \log n)$, where $\epsilon > 0$ is an arbitrarily small constant, such that the face of $\mathcal{A}(U)$ containing a query point is located in time $O(\log^3 n)$. If two query points are in the same cell of $\mathcal{A}(U)$, a collision-free path connecting them is produced. This result is obtained by exploiting powerful and so far overlooked properties of sparse nets introduced by Chazelle [*Discrete Comput. Geom.*, 9 (1993), pp. 145–158]. If the $(d-1)$-simplices in U have pairwise-disjoint interiors and $d \geq 3$, improved bounds are obtained. A data structure is described that uses $O(n^{d-1})$ storage and is built deterministically in time $O(n^{d-1})$ such that point-location queries are solved in time $O(\log n)$. Also, as a by-product, this method gives the first optimal worst-case algorithm for triangulating a nonsimple polyhedron in 3-space.

Key words. arrangements of simplices, point location, sparse nets, motion planning, triangulations

AMS subject classifications. 68P05, 68Q25, 68Q40, 68U05

1. Introduction.

1.1. Point location: Definition and previous results.

Point location is a central problem in computational geometry [PS85, Ede87, Meh84], and a continuous stream of research papers have been published on this topic beginning in the early days of computational geometry. Let U be a set of n possibly intersecting $(d-1)$-simplices in E^d. For example, we consider segments on the plane, triangles in 3-space, tetrahedra in 4-space, etc. Such a set U decomposes E^d into open cells of dimension d and relatively open cells of dimension k with $0 \leq k < d$. This collection of cells is called the *arrangement* $\mathcal{A}(U)$ induced by U in E^d. We associate a unique identifier to each cell in $\mathcal{A}(U)$ from a universe of distinct labels. Let K be the cardinality of $\mathcal{A}(U)$, also referred to as the combinatorial complexity of the arrangement of U. The point-location problem consists of preprocessing U into a data structure so that subsequently the following query can be answered efficiently:

(A) Given a query point q, find any cell in $\mathcal{A}(U)$ whose relative interior contains q.

In this paper, we also consider two variations of the point-location problem for which slightly different bounds are known.

(B) Given a query point q, determine whether q is incident to any element of U (incidence query).

(C) Given a query point q, determine the element of U immediately below q in a fixed direction. (This variation is also called vertical ray-shooting problem.)

* Received by the editors October 13, 1993; accepted for publication (in revised form) January 25, 1995. A preliminary version of this work appeared in *Proc. 26th ACM Symposium on Theory of Computing*. This research was supported by the CNR of Italy within the 1993–1994 Senior Visiting Scientist program at ICSI.

[†] Department of Computer Science, King's College, Strand, London WC2R 2LS, UK. Current address: Instituto di Matematica Computazionale, Consiglio Nazionale delle Richerche, 46 Via S. Maria, 56126 Pisa, Italy.

Several results are known when restrictions are placed on the input set U and on the dimension d. We assume that the dimension d is a small constant and the multiplicative constants in the "big-Oh" notation depend on d.

Hyperplanes. When U is a set of hyperplanes H in E^d for $d \geq 2$, Clarkson [Cla87] solves point-location queries in time $O(\log n)$ using a data structure of size[1] $O(n^{d+\epsilon})$ which is constructed in *expected* time $O(n^{d+\epsilon})$. Chazelle and Friedman [CF92] solve the vertical ray-shooting problem in $O(\log n)$ time using a data structure of size $O(n^d)$, but with an high preprocessing time. Chazelle [Cha93] solves point-location queries in time $O(\log n)$ using a data structure of size $O(n^d)$ built deterministically in time $O(n^d)$. Chazelle [Cha93] solves incidence queries in $O(\log n)$ time using a data structure of size $O(n^d/(\log n)^{d-1})$ which is built deterministically in time $O(n^d/(\log n)^{d-1})$. Matoušek [Mat93] solves vertical ray-shooting queries within the same bounds.

Planar maps. The point-location problem has been studied extensively for the case when U is a set of segments in E^2 forming a planar map (e.g., [LP77, Pre81, LT80]). Even a partial list of all the important results on this problem is beyond the scope of this paper, and for a more complete discussion of point-location algorithms for planar maps, we refer the reader to Preparata and Shamos [PS85] and Edelsbrunner [Ede87]. Point-location queries (variations (A), (B), and (C)) are answered in time $O(\log n)$ using $O(n)$ storage and $O(n \log n)$ preprocessing time (see, e.g., [Kir83, EGS86, Col86, ST86]). The techniques used in [Kir83, EGS86] are essentially based on properties of planar maps that do not hold for generalizations of such maps to three- and higher-dimensional spaces. The results in [Col86, ST86] are based on the sweeping paradigm. These algorithms sweep a line l on the plane and dynamically (or semidynamically) maintain the intersection of l with the cells in $\mathcal{A}(U)$. For the plane, the problem is to maintain dynamically a one-dimensional arrangement, which is relatively easy. Preparata and Tamassia [PT88] give an efficient method for dynamic maintenance of planar maps, which is used in [PT89] in connection with a plane sweeping in 3-space. The sweeping paradigm fails in dimension four since we do not have yet an efficient method for dynamic maintenance of three-dimensional maps.

Pairwise-interior-disjoint simplices in $d \geq 3$. If U is a set of n interior-disjoint triangles in E^3 forming a *convex arrangement*, the method of [PT89, GT91] solves point-location queries in time $O(\log^2 n)$ using $O(n \log n)$ storage and preprocessing time. If the set U is still formed by interior-disjoint triangles in E^3 but the arrangement of U is not composed of convex cells, Mulmuley [Mul91] gives a method that solves point-location queries in time $O(\log^2 n)$ using $O(n^2)$ storage and $O(n^2 \log n)$ preprocessing time.

Intersecting simplices. When U is a set of hyperplanes in general position or a set of interior-disjoint simplices, the size of the arrangement of U is bounded tightly by a function of n. Therefore, the upper bounds are expressed as a function of one variable n. When we allow the simplices in U to intersect each other, we have that K is in the range $[n, n^d]$. Therefore, we are interested in algorithms whose time and storage bounds depend explicitly on K. If U is a set of n possibly intersecting segments in the plane, we can use the optimal algorithm of Chazelle and Edelsbrunner [CE92] to construct $\mathcal{A}(U)$ in time $O(n \log n + K)$. Since $\mathcal{A}(U)$ is a planar map of size $O(K)$, we can use the point-location methods for planar maps and solve point-location problems

[1] Here and throughout the paper, we denote with ϵ an arbitrarily small positive real number. The multiplicative constant in the big-Oh notation may depend on ϵ and it goes to infinity as ϵ tends to 0.

in time $O(\log n)$ using $O(K)$ storage and $O(n \log n + K)$ preprocessing time.

Recently and independently, de Berg, Guibas, and Halperin [dBGH94] have extended the sweeping-plane technique in [Mul91] to sets of intersecting triangles in 3-space. They build a decomposition of $\mathcal{A}(U)$ into $O(n^{2+\epsilon} + K)$ convex cells using $O(n^{2+\epsilon} + K \log n)$ time. Point-location queries are answered in time $O(\log^2 n)$. Since the approach is based on a sweeping plane, it suffers of the limitations mentioned before for extensions to higher-dimensions.

1.2. New results on point location: Intersecting simplices. The main contribution of this paper is a method for solving point-location queries in an arrangement of n possibly intersecting simplices in E^d for $d \geq 2$. Our time and storage bounds depend explicitly on the number of cells in the arrangement of simplices $K = |\mathcal{A}(U)|$. We answer point-location queries in time $O(\log^3 n)$ using a data structure of size $O(n^{d-1+\epsilon} + K)$ which is built deterministically in time $O(n^{d-1+\epsilon} + K \log n)$. We answer incidence queries in time $O(\log^2 n)$ and vertical ray-shooting queries in time $O(\log^3 n)$ using $O(n^{d-1+\epsilon} + K)$ preprocessing time and storage.

Nonintersecting simplices. If the simplices in U are pairwise interior disjoint, we obtain improved time and storage bounds for $d \geq 3$. We give a deterministic algorithm that builds a data structure of size $O(n^{d-1})$ in time $O(n^{d-1})$. Point-location, vertical ray-shooting, and incidence queries are answered using this data structure in time $O(\log n)$.

1.3. Motion planning among simplices. The results in this paper are relevant for motion-planning problems. For background material on motion planning, we refer the reader to [SS90, Lat91]. The motion-planning problem in its simplest (purely geometrical) form is defined in [SS89] as follows:

> Let B be a robot system consisting of a collection of rigid subparts having a total of k degrees of freedom, and suppose that B is free to move in two- or three-dimensional space V amidst a collection of obstacles whose geometry is known to the robot system. The *motion-planning problem* for B is as follows: Given an initial position Z_1 and a desired final position Z_2 of B, determine whether there is a continuous obstacle-avoiding motion of B from Z_1 to Z_2, and if so, plan such a motion.

Since B has k degrees of freedom, a position of B is representable as a point in E^k. The subset of E^k representing positions in which B does not collide with obstacles in V is called the *space of free positions* $FP \subset E^k$. Thus the motion-planning problem is reduced to the problem of finding a path connecting Z_1 and Z_2 in FP. If FP is a semialgebraic set in E^k, Schwartz and Sharir [SS83] solve the motion-planning problem in time polynomial in the number of algebraic constraints defining FP and their maximal degree and doubly exponential in k. Canny [Can87] gives an algorithm with running time roughly $O(n^k)$, where n is the number of algebraic constraints.

Point robots. If the robot B is a point in d-dimensional space and the obstacles are simplices, we have the simplest d-dimensional motion-planning problem. Aronov and Sharir [AS92] give an $O(n^{d-1} \log n)$ upper bound on the complexity of one cell in an arrangement of n $(d-1)$-simplices in E^d. For $d = 3$, this bound and the method in [AS90] result in an algorithm for computing one cell of $\mathcal{A}(U)$ in $O(n^{2+\epsilon})$ expected time.

A recent result of de Berg, Matoušek, and Schwarzkopf proves that any two points connected by a collision-free path among n possibly intersecting *convex* obstacles in

E^d can also be connected by a piecewise-linear path composed of $O(n^{(d-1)\lfloor d/2+1 \rfloor})$ segments. Algorithms for computing such paths are provided only for $d = 2$.

1.4. New results in motion planning. The technique we develop for solving point-location queries among simplices in E^d gives us with little extra effort a method for planning motions of point robots among obstacles represented by simplices in E^d. Our result is relevant when the initial position is not fixed but rather part of the query, as is often the case during the design and simulation of robotic systems. In this case, it is more convenient to compute the whole arrangement of simplices once during preprocessing rather than computing single cells on-line. Moreover, since motion-planning problems for more general robots can be solved by reduction to motion planning for point robots in higher-dimensional space, we can use our method to compute the free space FP whenever the expanded obstacles in configuration space are simplices. The free space FP is, in general, only a subcomplex of $\mathcal{A}(U)$, and in the worst case, the two sets can have complexity differing by orders of magnitude. On the other hand, in more practical situations, the arrangement $\mathcal{A}(U)$ is likely to be sparse when the obstacles are far from each other in terms of the size of the robot B. Our algorithm is able to take advantage of the sparsity of $\mathcal{A}(U)$, and in such cases, it provides a more efficient algorithm than general roughly $O(n^k)$ solutions [Can87] or solutions based on extending the simplices into hyperplanes.

For example, we use our results to find the isotopy classes of lines induced by a set of polygons in 3-space which are on a family of parallel planes. We map lines in E^3 as points in E^4 and edges in E^3 as simplices in E^4. The arrangement $\mathcal{A}(U)$ of these simplices represents the isotopy classes of lines. Thus using $O(n^{3+\epsilon} + K)$ storage, we can find in time $O(\log^3 n)$ the isotopy class of a query line and produce collision-free motions of a line. In this context, K is related to the number of lines that meet four edges of the input polygons. This particular motion-planning problem arises in controlling a radiation beam for radiosurgery of the brain. The model of the human brain as a collection of parallel polygons can be obtained in a natural way by using a CAT scanner.

1.5. Triangulations. Often, point-location methods are based on decomposing the free space E^d/U into convex parts. Such decompositions are referred to generically as *triangulations.* The problem of building triangulations is important in its own right with a wide range of applications. Most of the known results are for $d = 2$ (see, e.g., [Cha87]) and fewer are for higher-dimensional spaces (i.e., $d \geq 3$). In this paper, we consider the problem in $d \geq 3$ when U is a set of pairwise-interior-disjoint $(d-1)$-simplices. This class of input is important because it includes boundaries of general polyhedra in E^d.

A concept similar to that of a triangulation is that of a *binary space partition* (BSP). Given a set U of n interior-disjoint triangles in 3-space, a BSP of E^3 induced by U (denoted BSP(U)) is a hierarchical partition of E^3 into *convex cells* associated with the nodes of a binary tree. The root is associated with the whole space E^3 and the regions associated with the children of node v form a convex partition of the region associated with v. The leaves are associated with regions whose interior does not meet any triangle in U. Paterson and Yao show in [PY90] how to obtain a BSP of size $O(n^2)$ in time $O(n^3)$ for $d = 3$. In higher-dimensional space, the method in [PY90] builds a BSP of size $O(n^{d-1})$ in time $O(n^{d+1})$.

For the special case when U is the boundary of a polyhedron in E^3 and U is connected, Chazelle [Cha84] gives a triangulation of size $O(n^2)$ which is built in time $O(n^2)$. Both bounds are worst-case optimal. The method in [Cha84] relies on the connectivity of the boundary, and therefore this method does not extend naturally to polyhedra with disconnected boundaries. Some triangulations for special polyhedral sets in E^3 are discussed in [Ber93].

1.6. New results on triangulations. In this paper, we show a method for building a triangulation induced by pairwise-interior-disjoint $(d-1)$-simplices in d-dimensional space $(d \geq 3)$. We obtain a *deterministic* algorithm that builds a triangulation of size $O(n^{d-1})$ in time $O(n^{d-1})$. To our knowledge, no better algorithm is known for dimension $d \geq 4$ for the case when U is a set of pairwise-interior-disjoint simplices even forming a connected boundary. The size and time bounds of our algorithm are worst-case optimal for $d = 3$, as follows from an $\Omega(n^2)$ lower bound in [Cha84]. For $d = 3$, our result is the first worst-case optimal $\Theta(n^2)$ algorithm for triangulating a nonsimple polyhedron in 3-space.

1.7. On cuttings and sparse nets. The main tools used in this paper are *cuttings* and *sparse nets*. Given n hyperplanes in E^d, a $(1/r)$-cutting is a collection of simplices with disjoint interiors which together cover E^d and such that the interior of any simplex is cut by at most n/r hyperplanes. Chazelle describes a deterministic algorithm to construct $(1/r)$-cuttings of size $O(r^d)$ in time $O(nr^{d-1})$ [Cha93]. This algorithm is the basis of improved methods for locating points in arrangements of hyperplanes and for many other geometric problems. Matoušek uses the algorithm in [Cha93] in order to improve time and storage bounds on the simplex range-searching problem [Mat92].

The algorithm for constructing $(1/r)$-cuttings relies on properties of the so-called *sparse nets*, which are special subsets of an input set H of hyperplanes. Intuitively, a sparse net R of a set H for a region s of E^d is a subset of H such that the number of vertices in the arrangement of H within s is well approximated by the number of vertices of the arrangement of R within s.

In [Cha93, Mat92], this property of sparse nets is applied to objects (hyperplanes) that produce "globally dense" arrangements (i.e., every d hyperplanes in general position meet in one point) by exploiting the "local sparsity" of the arrangements. Previous randomized (e.g., [Cla87]) and deterministic (e.g., [Mat91]) techniques are already quite efficient for globally dense arrangements, and therefore we gain polylogarithmic or small polynomial factors by using the cuttings described in [Cha93]. In this paper, we show that sparse nets are very useful computational tools for objects like $(d-1)$-simplices that produce "globally sparse" arrangements (i.e., not every d $(d-1)$-simplices meet, and in general there can be between 0 and $O(n^d)$ points meeting d $(d-1)$-simplices).

1.8. The method. We will show three data structures to solve incidence queries, vertical ray-shooting queries, and point-location queries. Each data structure uses the previous one. We first tackle the problem of solving incidence queries. The general strategy is a classical geometric divide-and-conquer. We select a subset $N \subset U$ and use N to partition E^d into convex cells of constant descriptive complexity (also referred to as *elementary* cells). We compute the simplices of U intersecting each elementary cell and apply the same method recursively within the elementary cell. At each recursive call, the size of the problem is reduced, and eventually we obtain elementary cells which do not intersect any simplex in U. At this point, we stop the construction.

The elementary cells generated during the preprocessing are organized in a search tree which has the flavour of a multidimensional extension of the *hereditary segment tree* [CEGS89, dBGH94] or of a multidimensional extension of search trees in the *trapezoid method* for planar point location [Pre81, PS85].

We answer the incidence query by visiting the tree until we either detect an incidence of the query point with a simplex in U or reach a leaf and conclude that there is no incidence.

The main contribution of this paper is in the analysis techniques that allow us to derive the input-sensitive bounds on the storage and time bounds. The difficult task is to make sure that we obtain at each level of the search tree a fine-tuned bound on the number of elementary cells. If we use a coarse bound on the number of elementary cells, we will easily end up with total storage $O(n^d)$, which falls short of our goal of a bound of roughly $O(K + o(n^d))$. This tight control on the size of the construction is attained by exploiting the properties of the sparse nets [Cha93].

Chazelle uses the sparse nets in order to produce hierarchical cuttings for sets of hyperplanes. These cuttings than are the main ingredient of his method for solving point-location queries in arrangements of hyperplanes [Cha93]. Here we follow the same general approach of [Cha93] with two main differences. Algorithmically, we construct elementary cells in d-space and also elementary cells in $(d-1)$-space, which are then "lifted" to d-space. In the analysis, we use finer arguments on the number of elementary cells in order to obtain a final bound that depends explicitly on K.

Once we have the data structure for incidence queries, we add auxiliary data structures that enable us to solve vertical ray-shooting queries within the same asymptotic bounds for preprocessing and storage.

We solve incidence queries and vertical ray-shooting queries using the fact that these are *decomposable* queries, that is, knowing the solution for sets of simplices U_1 and U_2, we can easily find the solution for $U_1 \cup U_2$. This does not hold for point-location queries among simplices. In fact, if we partition the input set U, we may lose essential connectivity information on the set $E^d \setminus U$. When the input is formed by hyperplanes, two points are in the same cell of the arrangement if and only if the two points are on the same side of each hyperplane. For simplices, this simple characterization no longer holds and we need more powerful methods to identify the cells in $\mathcal{A}(U)$.

Our method to overcome these difficulties consists of building a sequence of sets of graphs which encode the topological information. This sequence is indexed by the dimension of the subspaces in which our arrangement is contained, starting from dimension two up to dimension d. In this construction, ray-shooting operations are an essential ingredient needed to "lift" the topological information from lower-dimensional to higher-dimensional arrangements. At the end of the preprocessing phase, we have built the connectivity graph for $\mathcal{A}(U)$ and each face is properly labelled. Given a query point p, we use a sequence of d vertical ray-shooting queries to find a vertex of $\mathcal{A}(U)$ in the same face

When the set U is formed of pairwise-disjoint simplices, we use a more direct line of attack that aims at generating a hierarchical triangulation of the free space $E^d \setminus U$. The point-location procedure is a natural product of the hierarchical triangulation.

The paper is organized as follows. Section 2 recalls the main properties of the sparse nets [Cha93]. In §3, we give and analyze the algorithms for solving incidence queries and vertical ray-shooting queries. In §4, we give and analyze the algorithm for solving point-location queries. In §5, we give the algorithm for constructing a triangulation induced by disjoint simplices; as a corollary, we obtain a method for solving point-location queries for this particular type of input set.

2. Sparse nets and hierarchical cuttings. In this section, we give the basic definitions and lemmas from [Cha93] which form the background to the results shown in the subsequent sections. Let H be a set of n hyperplanes in the Euclidean d-dimensional space E^d. We assume that H is in *general position*, meaning that exactly d hyperplanes meet in a common point. Let $R \subseteq H$ be a subset of $\rho \leq n$ hyperplanes. For a segment e, let R_e (resp. H_e) be the number of hyperplanes in R (resp. H) intersecting e but not containing e. For a d-simplex s, let $v(R,s)$ (resp. $v(H,s)$) be the number of vertices of the arrangement created by R (resp. H) contained in the relative interior of s. We denote by $H(s)$ the subset of hyperplanes of H intersecting s. All definitions and lemmas in this section extend to improper simplices (i.e., with some vertices at infinity). Let r be a positive integer number.

DEFINITION 1. *R is a $(1/r)$-approximation for H if for any segment e,*

$$\left| \frac{R_e}{|R|} - \frac{H_e}{|H|} \right| < \frac{1}{r}.$$

DEFINITION 2. *R is a $(1/r)$-net for H if for any segment e, $H_e > n/r$ implies $R_e > 0$.*

DEFINITION 3. *R is a sparse $(1/r)$-net for (H,s) if for any segment e, $H_e > n/r$ implies $R_e > 0$ and the following inequality holds:*

$$v(R,s) \leq 4 \left(\frac{|R|}{|H|} \right)^d v(H,s).$$

DEFINITION 4. *A $(1/r)$-cutting for H is a partition of E^d into interior-disjoint simplices such that any simplex meets at most n/r hyperplanes of H. The number of simplices in the partition is called the size of the cutting.*

Let us suppose that we are given a set H of n hyperplanes in E^d and a collection Θ of d-simplices which partitions of E^d. Let s be one of these d-simplices in Θ, r_0 be a constant, and $i > 0$ be an integer. Assume that for any s in Θ, we have $|H(s)| \leq n/r_0^{i-1}$. Given these conditions, Θ is a $(1/r_0^{i-1})$-cutting for H. The objective is to locally refine each simplex s in order to obtain a $1/r_0^i$-cutting for H. If it happens that $|H(s)| \leq n/r_0^i$, then there is no work to be done since s satisfies the condition for being an element of a $(1/r_0^i)$-cutting.

Therefore, we consider the case when $n/r_0^i \leq H(s) \leq n/r_0^{i-1}$. Computing a sparse net for $(H(s), s)$ directly is time consuming; therefore, following Chazelle [Cha93], we first generate an approximation $A(s)$ of $H(s)$ and then build a strong net for $(A(s), s)$, whose triangulation is the desired refinement for s. We define $\rho_0 = r_0^i |H(s)|/n$ and $\rho = \rho_0 \log \rho_0$. We can achieve our aim with a sparse net of size ρ. The following lemmas summarize important properties of the construction in [Cha93].

LEMMA 1 (see [Cha93]). *Let $A(s)$ be a $(1/(2d\rho_0))$-approximation of $H(s)$ and $R(s)$ a sparse $(1/(2d\rho_0))$-net for $A(s)$ in s; then the following inequality holds:*

$$v(R,s) \leq 4(\rho/|H(s)|)^d v(H,s) + 4\rho^d/\rho_0.$$

LEMMA 2 (see [Cha93]). *A canonical triangulation of the sparse net $R(s)$ of Lemma 1 has size $O(\rho^{d-1} + v(R,s))$.*

Observation. Combining the bounds of Lemmas 1 and 2, we obtain a bound on the number of simplices in the refinement of s:

$$O\left(\rho_0^{d-1} \log^d \rho_0 + \left(\frac{\rho^d}{H(s)^d} \right) v(H,s) \right).$$

LEMMA 3 (see [Cha93]). *The approximation $A(s)$ and the sparse net $R(s)$ of Lemma 1 are computed in time $O(|H(s)|)$.*

LEMMA 4 (see [Cha93]). *The number of hyperplanes in H intersecting any simplex in the canonical triangulation of Lemma 2 is less than or equal to n/r_0^i.*

The collection of all the simplices obtained as a result of canonical triangulations of the sparse nets $R(s)$ of Lemma 1 within s, together with those simplices $s \in \Theta$ for which no work was done, for all $s \in \Theta$, is a $(1/r_0^i)$-cutting for H.

3. Incidence and vertical ray-shooting queries.

3.1. Preliminary definitions. We fix once and for all a vertical direction in E^d. The following definitions are essential for the algorithm.

DEFINITION 5. *A d-cell in E^d is a d-dimensional simplex in E^d or an infinite prism whose axis is vertical and whose cross-section is a $(d-1)$-simplex.*

DEFINITION 6. *A $(d-1)$-simplex t partially covers a d-cell s if t intersects s and the vertical projection of t does not completely contain the vertical projection of s.*

If $(d-1)$-simplex t partially covers a d-cell s, then the vertical projection of some $(d-2)$-face of t will intersect the vertical projection of s.

DEFINITION 7. *A $(d-1)$-simplex t completely covers a d-cell s if t intersects s and the projection of t contains the projection of s.*

The concept of partially covering and totally covering $(d-1)$-simplices can be seen as an extension to higher dimensions of the concept of "short" and "long" segments in hereditary segment trees. In this paper, d-cells will be called also *elementary cells.*

3.2. A data structure for incidence queries. We are given a set U of n possibly intersecting $(d-1)$-simplices in E^d. We build a sequence of sets of d-cells $C_0, \ldots, C_l$, where $l = \log_{r_0} n$ and r_0 is a suitable constant. The base of the construction is C_0 and the construction precedes inductively from C_{i-1} to C_i. The set C_i is a collection of d-cells with auxiliary information $(s, P(s), Q(s))$, where s is d-cell in E^d, $P(s)$ is a subset of simplices in U *partially covering* s, and $Q(s)$ is a subset of simplices in U *covering* s. In each set C_i, the union of the elementary cells is E^d. The invariants maintained over the sets C_i are

(I) $P(s)$ empty or $|P(s)| \leq n/r_0^i$,

(II) $|Q(s)| \leq r_0 n/r_0^i$.

3.3. The incremental step of the algorithm. The first set is $C_0 = \{(E^d, U, \emptyset)\}$, which satisfies invariants (I) and (II). The algorithm to construct C_k from C_{k-1} for $k > 0$ works in two main phases.

1. For any elementary cell s in C_{k-1}, if $|Q(s)| \leq r_0 n/r_0^k$, then we add to C_k the triple $(s, \emptyset, Q(s))$. Otherwise, by the induction hypothesis, we have that $r_0 n/r_0^k < |Q(s)| \leq r_0 n/r_0^{k-1}$ and we have the conditions for the application of Lemma 1. We build a sparse net of Lemma 1 in s on $Q(s)$. We triangulate the sparse net, thus obtaining a set of d-cells $(\sigma, \emptyset, Q(\sigma))$, where $Q(\sigma)$ is the subset of elements of $Q(s)$ totally covering σ.

We choose the parameter of the net $\rho_0'(s) = r_0^{k-1}|Q(s)|/n$ and thus obtain $Q|(\sigma)| \leq r_0 n/r_0^k$ for each new cell σ as follows from Lemma 4. Note that for every s in C_{k-1}, we have $\rho_0'(s) \leq r_0$, which is an important property exploited in the analysis.

2. For any s in C_{k-1}, if $P(s)$ is empty, we skip phase 2. If $|P(s)| \leq n/r_0^k$, then we add to C_k the triple $(s, P(s), \emptyset)$. Otherwise, by the induction hypothesis, we have $n/r_0^k < |P(s)| < n/r_0^{k-1}$. We vertically project the $(d-2)$-faces of simplices in $P(s)$

onto a $(d-1)$-dimensional subspace and extend the projections into full hyperplanes in this subspace. We denote this set of hyperplanes in $(d-1)$-space with $P'(s)$. We vertically project s and decompose the projection into (a constant number of) $(d-1)$-simplices. For any such region s', we make a sparse net of Lemma 1 for the hyperplanes $P'(s)$ in s'. We triangulate the sparse net and obtain a set of $(d-1)$-cells. We extend the $(d-1)$-cells vertically in d-space within s, obtaining infinite prisms $(\eta, P(\eta), Q(\eta))$, where $P(\eta)$ is the subset of simplices in $P(s)$ partially covering η and $Q(\eta)$ is the subset of simplices in $P(s)$ totally covering η.

We choose the parameter of the nets $\rho_0''(s) = r_0^k|P(s)|/n$ in order to obtain $|P(\eta)| \leq n/r_0^k$ from Lemma 4 and $|Q(\eta)| \leq |P(s)| \leq r_0 n/r_0^k$ from the induction hypothesis. Note that, for every s in C_{k-1} we have $\rho_0''(s) \leq r_0$.

The collection of all the elementary cells generated in the two phases is C_k. The elementary cells in C_k satisfy invariants (I) and (II). The d-cells produced in the two phases are organized in a two-level search tree. The search trees on d-cells built in phase 1 (resp. 2) will be called Q-trees (resp. P-trees). The search tree built on the sequence $C_0, \dots, C_l$ is used as sketched in the introduction to test whether a query point q is incident to any simplex in U. The main issues are the time spent in the construction and the size of the search tree.

3.4. Analysis of the algorithm. To simplify the notation, we set $\rho_0(s) = \max\{\rho_0'(s), \rho_0''(s)\}$. We denote by K the total number of vertices in the arrangement of the set of simplices U which are incident to at least d of the $(d-1)$ simplices in U; clearly, we have $K \leq \binom{n}{d}$. We denote by $c_1, c_2, \dots$ absolute multiplicative constants which may depend on d. We obtain a bound on $|C_k|$ by summing the number of new d-cells generated in one iteration of the algorithm for all the d-cells in C_{k-1}. We use Lemma 2 to bound the number of d-cells obtained at each iteration of the algorithm.

$$
(1) \qquad \begin{aligned}
|C_k| \leq & \sum_{s \in C_{k-1}} c_1[\rho_0^{d-1}(s) \log^{d-1} \rho_0(s) + v(R(s), s)] \\
& + \sum_{s \in C_{k-1}} c_2[\rho_0^{d-2}(s) \log^{d-2} \rho_0(s) + v(R'(s), s')].
\end{aligned}
$$

We apply the observation after Lemma 2, the observation that $\rho(s) \leq r_0$ always, and the induction hypothesis on the size of $Q(s)$ and $P(s)$, thus obtaining a new expression bounding $|C_k|$:

$$
(2) \qquad \begin{aligned}
& \sum_{s \in C_{k-1}} c_3[r_0^{d-1} \log^d r_0 + (r_0^k \log r_0/n)^d v(Q(s), s)] \\
& + \sum_{s \in C_{k-1}} c_4[r_0^{d-2} \log^{d-1} r_0 + (r_0^k \log r_0/n)^{d-1} v(P'(s), s')].
\end{aligned}
$$

We rearrange the summations and subsume smaller terms under larger ones:

$$
(3) \quad \sum_{s \in C_{k-1}} c_5[r_0^{d-1} \log^d r_0 + (r_0^k \log r_0/n)^d v(Q(s), s) + (r_0^k \log r_0/n)^{d-1} v(P'(s), s'))].
$$

Finally, noticing that $\sum_{s \in C_{k-1}} v(P'(s), s') = O(n^{d-1})$, we obtain the following recursive inequality in the variable k:

$$
(4) \quad |C_k| \leq c_6[r_0^{d-1}(\log^d r_0)|C_{k-1}| + r_0^{kd}(\log^d r_0)(K/n^d) + r_0^{k(d-1)}(\log^{d-1} r_0)].
$$

In the next lemma, we solve recursion (4) and find a bound on $|C_k|$ in terms of n and K.

LEMMA 5. *$|C_k| \leq Dr_0^{k(d-1+\epsilon)} + F(K/n^d)r_0^{kd}$, where F and D are constants with respect to k, n, and K.*

Proof. Let us define a value $\tilde{k}$ such that

$$\left(K/\binom{n}{d}\right) r_0^{kd} = r_0^{k(d-1+\epsilon)}. \tag{5}$$

That is,

$$\tilde{k} = (1/(1-\epsilon)) \log_{r_0} \left(\binom{n}{d}/K\right).$$

Let $\bar{k} = \lfloor \tilde{k} \rfloor$. Note that for $k \leq \bar{k}$,

$$\left(K/\binom{n}{d}\right) r_0^{kd} \leq r_0^{k(d-1+\epsilon)}, \tag{6}$$

while for $k > \bar{k}$,

$$\left(K/\binom{n}{d}\right) r_0^{kd} \geq r_0^{k(d-1+\epsilon)}. \tag{7}$$

We will show that for $k \leq \bar{k}$, equation (4) has a solution $|C_k| \leq Dr_0^{k(d-1+\epsilon)}$ for a constant D. We will show that for $k > \bar{k}$, equation (4) has a solution $|C_k| \leq F(K/n^d)r_0^{kd}$ for a constant F. Notice that equation (4) is a recursive equation on a function of k; thus the term K/n^d is treated like a constant during the solution of the recurrence.

Case 1: $0 \leq k \leq \bar{k}$. We prove the bound $|C_k| \leq Dr_0^{k(d-1+\epsilon)}$ by induction. For $k = 0$, the bound is trivially true since $C_0 = (E^d, U, \emptyset)$ and $|C_0| = 1$. Otherwise, we inductively assume that the bound holds for C_{k-1}:

$$\begin{aligned}
&|C_k| \\
&\leq c_6[r_0^{d-1+\epsilon}(\log^d r_0)D(1/r_0^\epsilon)r_0^{(k-1)(d-1+\epsilon)} + r_0^{kd}(\log^d r_0)(K/n^d) + r_0^{k(d-1)}(\log^{d-1} r_0)] \\
&\leq c_6[(\log^d r_0)D(1/r_0^\epsilon)r_0^{k(d-1+\epsilon)} + r_0^{kd}(\log^d r_0)(K/n^d) + r_0^{k(d-1)}(\log^{d-1} r_0)].
\end{aligned}$$

Since $k \leq \bar{k}$, we have $(K/n^d)r_0^{kd} = O(r_0^{k(d-1+\epsilon)})$. Thus

$$|C_k| \leq c_7[(\log^d r_0)D(1/r_0^\epsilon) + (\log^d r_0) + (\log^{d-1} r_0)]r_0^{k(d-1+\epsilon)}. \tag{8}$$

Choosing r_0 dependent on ϵ and choosing D large enough, we can make sure that

$$c_7[(\log^d r_0)D(1/r_0^\epsilon) + (\log^d r_0) + (\log^{d-1} r_0)] \leq D,$$

and therefore we prove the bound.

Case 2: $k > \bar{k}$. We prove by induction a bound $|C_k| \leq F(K/n^d)r_0^{kd}$. The base case is for $k = \bar{k}+1$. By using the bound of Case 1 for $C_{\bar{k}}$ within equation (4), we easily obtain a bound

$$\begin{aligned}
|C_{\bar{k}+1}| &\leq c_6[(\log^d r_0)Dr_0^{d-1}r_0^{\bar{k}(d-1+\epsilon)} + r_0^{\bar{k}d}(\log^d r_0)(K/n^d) + r_0^{\bar{k}(d-1)}(\log^{d-1} r_0)] \\
&\leq F(K/n^d)r_0^{d(\bar{k}+1)}
\end{aligned}$$

for a constant F depending on r_0, D, and ϵ. Let us assume that the bound holds inductively:

$$\begin{aligned}|C_k| &\le c_6[r_0^{d-1}(\log^d r_0)|C_{k-1}| + r_0^{kd}(\log^d r_0)(K/n^d) + r_0^{k(d-1)}(\log^{d-1} r_0)]\\ &\le c_6[r_0^d(\log^d r_0)F(1/r_0)(K/n^d)r_0^{(k-1)d} + r_0^{kd}(\log^d r_0)(K/n^d) + r_0^{k(d-1)}(\log^{d-1} r_0)].\end{aligned}$$

Since $k > \bar{k}$, inequality (7) holds and we have $r_0^{k(d-1)} < r_0^{k(d-1+\epsilon)} = O((K/n^d)r_0^{kd})$. Therefore,

$$(9) \qquad |C_k| \le c_8[(\log^d r_0)F(1/r_0) + (\log^d r_0) + (\log^{d-1} r_0)](K/n^d)r_0^{kd}.$$

Choosing F and r_0 large enough, we have

$$c_8[(\log^d r_0)F(1/r_0) + (\log^d r_0) + (\log^{d-1} r_0)] \le F,$$

and the bound is proved. □

3.4.1. Storage and preprocessing time bounds. We continue the construction until k reaches the value $l = \log_{r_0} n$. The total number of elementary cells is

$$(10) \qquad \sum_{k=0}^{l} |C_k| \le \sum_{k=0}^{l} [Dr_0^{k(d-1+\epsilon)} + F(K/n^d)r_0^{kd}].$$

This is a summation of geometric series of ratio $r_0^{d-1+\epsilon}$ and r_0^d, which is proportional to the last term of the series. Thus we have a bound $O(Dr_0^{l(d-1+\epsilon)} + 2F(K/n^d)r_0^{ld}) = O(n^{d-1+\epsilon} + K)$ on the size of the search tree. The time spent on each d-cell in the search tree is linear in the number of $(d-1)$-simplices intersecting it by Lemma 3. Thus we have a bound

$$(11) \qquad \sum_{k=0}^{l} (nr_0/r_0^k)|C_k| \le \sum_{k=0}^{l} (nr_0/r_0^k)[Dr_0^{k(d-1+\epsilon)} + F(K/n^d)r_0^{kd}].$$

By simple manipulations and using the previous observation on the sum of geometric series, we obtain an upper bound $O((nr_0)[Dr_0^{l(d-2+\epsilon)} + F(K/n^d)r_0^{l(d-1)}])$, which is $O(n^{d-1+\epsilon} + K)$.

3.4.2. Query time for incidence queries. We organize the sets $C_0, C_1, \dots, C_l$ as levels of a two-layer search tree where the P-tree is the primary structure and the Q-tree is the secondary one. Given the query point q, we exhaustively search an elementary cell s at the root of our search-data structure, which contains q in its closure. Then we continue the search recursively in the P-tree and the Q-tree rooted at s. On the Q-tree, the search visits a single path of logarithmic depth. Therefore, we have a total cost $O(\log n)$ for searching a Q-tree. The query on a P-tree is solved recursively. If $T(h)$ is the time needed to solve the query starting from a node at height h, we have that

$$\begin{aligned}T(h) &\le O(\log n) + T(h-1),\\ T(1) &= O(1).\end{aligned}$$

Thus $T(\log n) = O(\log^2 n)$.

Since the total number of cells of any dimension in the arrangement $\mathcal{A}(U)$ is $O(n^{d-1} + K)$, we have the following theorem.

THEOREM 1. *Given a set U of n possibly intersecting $(d-1)$-simplices in d-space, we can build a data structure of size $O(n^{d-1+\epsilon} + K)$ deterministically in time $O(n^{d-1+\epsilon} + K)$, where K is the complexity of the arrangement $\mathcal{A}(U)$. Afterwards, we can determine in time $O(\log^2 n)$ whether a query point is incident to a simplex in U.*

3.5. Vertical ray shooting in arrangement of simplices. In order to solve the vertical ray-shooting problem, we add auxiliary data structures to the search tree built previously. Let us discuss Q-trees first. We consider cells $s \in C_{k-1}$ and $\sigma \in C_k$ such that $\sigma \subset s$ and σ is obtained by phase 2 of the algorithm. We determine during preprocessing the subset of $(d-1)$-simplices in $Q(s)/Q(\sigma)$ which lies below σ. We can extend the simplices of this subset into full hyperplanes and build a data structure for general ray shooting in convex polytopes of size roughly $O(|Q(s)|^{\lfloor d/2 \rfloor})$ [Sch92].[2] This data structure is associated with σ.

Let us now consider the P-trees. We consider cells $s \in C_{k-1}$ and $\sigma \in C_k$ such that $\sigma \subset s$ and σ is obtained by phase 1 of the algorithm. Note that in this case, both s and σ are vertical infinite cylinders. From this, it follows that if a query point q is in σ, then the simplex in $P(s)$ immediately below q is either the simplex in $P(\sigma)$ immediately below q or the simplex in $Q(\sigma)$ immediately below q. Vertical ray-shooting queries are solved by recursive calls on P-trees and by using the auxiliary data structures on Q-trees. On Q-trees, the search path visits $O(\log n)$ nodes, and at each node, $O(\log n)$ time is spent querying the data structure in [Sch92].

The query time for the recursion on the Q-trees rooted at a simplex s has an additional logarithmic factor; thus we obtain a time $O(\log^2 n)$ for one partial query. The query time satisfies the same equation as before with an additional $O(\log^2 n)$ term due to the queries on the auxiliary data structures. If $T(h)$ is the time needed to solve the query starting from a node at height h, we have that

$$\begin{aligned} T(h) &\leq O(\log^2 n) + T(h-1), \\ T(1) &= O(1). \end{aligned}$$

Thus $T(\log n) = O(\log^3 n)$. The time needed to construct the data structure is bounded by

$$\begin{aligned} \sum_{k=0}^{l} (nr_0^2/r_0^k)^{\lfloor d/2 \rfloor} |C_k| &\leq \sum_{k=0}^{l} (nr_0^2/r_0^k)^{\lfloor d/2 \rfloor} [Dr_0^{k(d-1+\epsilon)} + F(K/n^d) r_0^{kd}] \\ &\leq 2(nr_0^2)^{\lfloor d/2 \rfloor} [Dr_0^{l(d-1-\lfloor d/2 \rfloor+\epsilon)} + F(K/n^d) r_0^{l(d-\lfloor d/2 \rfloor)}] \\ &\leq 2r_0^d (Dn^{d-1+\epsilon} + FK) = O(n^{d-1+\epsilon} + K). \end{aligned}$$

The discussion above proves the following theorem.

THEOREM 2. *Given a set U of n possibly intersecting $(d-1)$-simplices in d-space, we can build a data structure of size $O(n^{d-1+\epsilon} + K)$ deterministically in time $O(n^{d-1+\epsilon} + K)$, where K is the complexity of the arrangement $\mathcal{A}(U)$. Afterwards, we can solve vertical ray-shooting queries in time $O(\log^3 n)$.*

[2] We could use the method in [AM92], but in doing so, we would add an extra logarithmic factor to the query time.

4. Point location among simplices. In this section, we tackle the general point-location problem by using the solution to the vertical ray-shooting problem as a subroutine. We encode the facial structure of $\mathcal{A}(U)$ in a sequence of sets of graphs. The final graph will have nodes for each face of $\mathcal{A}(U)$, and each node will have a label indicating the d-dimensional cell that has this face at its boundary. For a query point p, we retrieve at query time the labels associated with the faces containing it.

In the following discussion, we will have to deal with the fact that a face is incident to more than one cell, and therefore we must consider the role of the several "sides" of a face. The main idea is to walk along the boundary of cells and describe such a boundary as an incidence graph of lower-dimensional cells. We use ray shooting in order to join "islands" of the boundary of a cell that cannot be reached by the simple walking procedure. If we think of this problem on the plane with a set of segments, it is clear that we can describe the boundary of a cell by walking clockwise on the connected parts of its boundary. What follows is a generalization of this walking and jumping idea to higher-dimensional spaces.

4.1. Definitions. First, we give additional definitions that lay the groundwork for the main result. The terminology and the concepts used are consistent with those introduced in [AS92].

A set U of n $(d-1)$-simplices in E^d decomposes E^d into open cells of dimension d (also called d-faces) and into relatively open cells of dimension k for $0 \leq k < d$. Let $\mathcal{A}(U)$ be the collection of cells of any dimension induced by U. We assume that the simplices in U are in general position, meaning that any k of them (for $k = 2, \ldots, d$) intersect, if at all, in a polytope of dimension at most $d-k$. This intersection is called a $(d-k)$-*flap* in [AS92]. We assume that $d+1$ simplices in U do not have a point in common. We also assume that the intersection of any collection of k open faces of dimensions $i_1, \ldots, i_k$ is either empty or a polytope of dimension $d - \sum_j (d - i_j)$. A perturbation argument shows that the maximum size of the quantities of interest is attained by a set U in general position. We distinguish *outer k-faces*, which are contained in the relative boundary of a simplex in U, and *inner k-faces*, which are contained exactly in the interior of $d-k$ simplices and avoid their relative boundary. We distinguish between different sides of an (inner or outer) face. Let f be an inner k-face contained in the relative interior of $d-k$ simplices. The hyperplanes spanning these simplices divide space into 2^{d-k} open regions. A *side* of f is a pair (f, r), where r is one of these regions. If f is a d-cell, then it has only one side, namely (f, E^d). Notice that a $(d-1)$-face has two sides. Similar definitions hold for the outer k-faces with the difference that outer k-faces are contained in fewer than $d-k$ simplices.

DEFINITION 8. *A side (f, r) is a k-border of a d-cell C if either $f = C$ and $k = d$ or f is k-face on the boundary of C and some open neighborhood of f in $r \cup f$ is contained in $C \cup f$.*

Intuitively, the concept of k-border captures the fact that f is on the boundary of a d-cell C and that C is on the same side of f as r.

DEFINITION 9. *For $0 \leq k \leq i < d$, a (k, i)-border of C is a pair $((f, R), (g, Q))$ of borders of C of dimensions k and i such that $f \subset \bar{g}$ and $R \subset Q$.*

Let $G(U)$ be a graph whose nodes are k-borders of $\mathcal{A}(U)$ for all values of k. Two nodes (f, R) and (g, Q) are connected by an edge in $G(U)$ if $((f, R), (g, Q))$ is a (k, i)-border. We aim at generating a labelling of the nodes of $G(U)$ such that all the k-borders of a same d-cell C have the same label and this label is different from the label of k-borders of d-cell C' if $C \neq C'$.

4.2. The preprocessing algorithm. The construction of the labelled graph $G(U)$ for U in E^d is done in phases which are denoted with an index $j = 2, \dots, d$. At the end of phase j, we have a set of labelled incidence graphs, where each graph corresponds to an arrangement of simplices in dimension j. We use this information to build a set of graphs of arrangements in dimension $j + 1$. The output of the last phase for $j = d$ represents our labelled graph $G(U)$. To start our construction, let us consider dimension $j = 2$, which is our base case. We compute the collection $\mathcal{B}_2 = \binom{U}{d-2}$ of all of $(d - 2)$-subsets of U. Let β be an element of $\mathcal{B}_2$ and $q(\beta)$ be the 2-flap which is the intersection of the simplices in β. If $q(\beta)$ is not empty, we build the planar arrangement of the segments $U_\beta = \{s \cap q(\beta) | s \in U/\beta\}$ on $q(\beta)$. Using the method of [CE92], we can explicitly construct such in arrangement in time $O(n \log n + K_\beta)$, where K_β is the number of vertices in $\mathcal{A}(U_\beta)$. Within the same time, we can compute the graph $G(U_\beta)$ and label its k-borders with the name of the bordering cell. Moreover, we can add standard planar point-location data structures in order to be able to locate a 0-border of the 2-cell containing a query point.

We have established the result for $j = 2$. Now we assume $j > 2$ and assume inductively that we have computed the relevant data structure for $(j - 1)$. More precisely, we have the labelled graphs $G(U_\beta)$ for all $\beta \in \mathcal{B}_{j-1} = \binom{U}{d-(j-1)}$ and also the point-location/vertical ray-shooting data structures for all sets U_β. We shall use this information to construct the graphs $G(U_\gamma)$, where $\gamma \in \Gamma = \mathcal{B}_j = \binom{U}{d-j}$. Consider a particular $\gamma \in \Gamma$. We aim to construct the incidence graph over the set of simplices $U_\gamma = \{s \cap q(\gamma) | s \in U/\gamma\}$ on $q(\gamma)$. The relative boundary of $q(\gamma)$ is formed by $j - 1$ simplices, and we can assume inductively that we know the arrangement induced by U on the facets of $q(\gamma)$ and the corresponding incidence graph G.

An inner k-face f in $\mathcal{A}(U)$ is contained in $d - k$ simplices of U. We can split these $d - k$ simplices into two groups: $d - j$ of them define a flat $q(\gamma)$ to which f belongs; the $j - k$ remaining simplices define the affine span of f within $q(\gamma)$. We denote by $\Gamma(f) = \{\gamma \in \Gamma | f \subset q(\gamma)\}$ the subset of elements of Γ containing f in their corresponding j-flap. Similarly, we define $\mathcal{B}(f) = \{\beta \in \mathcal{B}_{j-1} | f \subset q(\beta)\}$. From the previous observation, $|\Gamma(f)| = \binom{d-k}{d-j}$ and $|\mathcal{B}(f)| = \binom{d-k}{d-(j-1)}$. Fixing a $\gamma \in \Gamma(f)$; is equivalent to fixing $d - j$ simplices incident to f; thus we can choose any one of the remaining $j - k$ simplices incident to f to define a set β in $\mathcal{B}(f)$. We therefore have the following set:

$$\mathcal{B}(f, \gamma) = \{\beta \in \mathcal{B}(f) | q(\beta) \subset q(\gamma)\},$$

representing the β's which contribute to γ. The cardinality of this set is $|\mathcal{B}(f, \gamma)| = j - k$.

The main idea is to use the regions of the k-borders in graphs in $G(\mathcal{B})$ to build the regions of k-borders in graphs in $G(\Gamma)$. First, we must establish with a counting argument that we indeed have enough nodes from lower-dimensional graphs to carry out the construction. One face f in $\mathcal{A}(U_\gamma)$ corresponds to 2^{j-k} k-borders in $G(U_\gamma)$, and each such k-border has associated a region delimited by $(j - k)$ facets. These facets are regions of k-borders of f in the set of graphs $G(U_\beta)$ for $\beta \in \mathcal{B}(f, \gamma)$. In order to construct $G(U_\gamma)$ for all $\gamma \in \Gamma$, we proceed in the following way: we duplicate every node in every G_β for $\beta \in \mathcal{B}$. This operation reflects the fact that every $(j - 1)$-cell has two sides in the new space $q(\gamma)$. Consequently, every label is split into a positive and a negative label. The total number of k-borders of a face f in graphs $G(U_\beta)$ for $\beta \in \mathcal{B}(f, \gamma)$ becomes

$$2[2^{j-1-k}(j - k)] = (j - k)[2^{j-k}].$$

Now the algorithm is easy: we build the regions of k-borders of f in $G(U_\gamma)$ by taking the regions of k-borders of f in $G(U_\beta)$ for $\beta \in \mathcal{B}(f, \gamma)$ as *facets*. We associate to the nodes in $G(U_\gamma)$ an initial labelling using the $(j - k)$ labels of the facets of the region of each k-border in $G(U_\gamma)$. We recall that the labels we are using are just names of $(j - 1)$-cells. At this stage, we have an initial labelled graph $G^i(U_\gamma)$ with a labelling consistent with the information we obtain from lower-dimensional graphs. On the other hand, this initial labelling is such that a single cell in $\mathcal{A}(U_\gamma)$ does not yet have a unique identifier. The next phase of the algorithm finds a unique identifier for every j-cell of $\mathcal{A}(U_\gamma)$ and updates the labels of the k-borders in $G(U_\gamma)$ correspondingly. Let C be a j-cell in $\mathcal{A}(U_\gamma)$.

Relabelling Step I. We build a label graph $LG(\gamma)$ whose nodes are nodes of $G(U_\gamma)$. We place an edge between two nodes of $LG(\gamma)$ if and only if the two initial labels of the nodes have no empty intersection. Then we run a standard connected-component algorithm on $LG(\gamma)$ [AHU74]. During the visit of the graph $LG(\gamma)$, we also compute for each connected component of the graph the lowest point in the vertical direction of the space $q(\gamma)$. At the end of this phase, every path-connected component of the boundary of C has a unique label.

Relabelling Step II. Now we proceed to identify the several connected components of the boundary of a j-cell C on $q(\gamma)$. We build in $q(\gamma)$ for U_γ the vertical ray-shooting data structure described in §3. We build a a graph of boundary components $BCG(\gamma)$ whose nodes are the connected components of the graph $LG(\gamma)$. We connect with an arc two nodes u and u' in $BCG(\gamma)$ if, shooting in $q(\gamma)$ from the lowest point of u, we reach a point on u'.

Afterwards, we find the connected components of $BCG(\gamma)$ using standard graph-searching algorithms. The next lemma states that the final labelling obtained from the connected components of the graph $BCG(\gamma)$ satisfies the uniqueness property.

LEMMA 6. *The identifiers of the connected components of $BCG(\gamma)$ are unique labels for the j-cells in $\mathcal{A}(U_\gamma)$.*

Proof. In order to simplify the argument, we assume that all the simplices in U are bounded, and we consider a solid compact d-ball B containing U in its interior. Let C be a nonempty j-cell of $\mathcal{A}(U_\gamma) \cap B$ and $B_1, \dots, B_k$ be the connected components of the boundary of C. Cell C is bounded and, in particular, at least one boundary component is a regular boundary as defined in [Ale56, p. 41], i.e., a set which separates E^d into two domains. Moreover, one of these two domains is bounded and the other is unbounded. The procedure for linking nodes in the graph $BCG(\gamma)$ produces a partial order among the nodes $B_1, \dots, B_k$. We show that there cannot be two bottom elements in this partial order.

If B_i is not a regular boundary, then the vertical ray from the lowest point of B_i is within C and will intersect another boundary component. Therefore, B_i cannot be a bottom element of the partial order. Let us assume by contradiction that we have two regular boundaries B_1 and B_2 with associated bounded domains D_1 and D_2 such that B_1 and B_2 are both bottom elements of the partial order. If this is the case, then the vertical *downward* ray from the lowest point of B_1 (resp. B_2) is in the unbounded domain defined by B_1 (resp. B_2). Therefore, we have that $C \subseteq D_1$ and $C \subseteq D_2$. Now we have two main cases to consider since the other cases lead easily to contradictions.

Case 1: $D_2 \subset D_1$, *from which* $C \subseteq D_2 \subset D_1$. But then B_1 cannot be on the boundary of C unless it is a subset of B_2. Since B_1 and B_2 are disjoint, we have a contradiction.

Case 2: $D_2 \subset E^d \setminus D_1$. In this case, $D_1 \cap D_2 = \emptyset$ and therefore $C = \emptyset$, which is a contradiction.

From the discussion above, it follows that all the boundary components of a cell C are included in the same component of the graph $BCG(\gamma)$.

Now we assume that B_1 is a boundary component of C_1, B_2 is a boundary component of C_2, and (B_1, B_2) is an arc in $BCG(U_\gamma)$. If this is the case, there is a vertical segment which connects B_1 and B_2, which does not intersect any simplex in U, and which at the extremes belongs to C_1 and C_2. It follows that C_1 and C_2 are path connected and therefore are the same j-cell of $\mathcal{A}(U_\gamma)$. We have proved that if the nodes of $BCG(\gamma)$ are the boundary-connected components of cells in $\mathcal{A}(U_\gamma)$, then the connected components of $BCG(\gamma)$ are in 1–1 correspondence with j-cells of $\mathcal{A}(U_\gamma)$.

Now let p_1 and p_2 be two points on a connected component B of the boundary of a j-cell C. Then there is a continuous path contained in B and in C that has p_1 and p_2 as extremes. We can reduce this path so that it visits each $(j-1)$-face of B at most once. Let $F_1, \ldots, F_k$ be the sequence of $(j-1)$-faces of a path connecting p_1 and p_2, and let f_i be the $(j-2)$-face shared by F_i and F_{i+1}. Assuming by induction on the dimension that we have a unique label for the faces F_i, we have that the initial labelling of the $(j-2)$-border f_i of C is (F_i, F_{i+1}). Therefore, if we build the graph $LG(\gamma)$, there will be a path connecting F_1 and F_k in $LG(\gamma)$.

On the other hand, if p_1 on face F_1 and p_2 on face F_2 belong to different boundary components B_1 and B_2, then any path between B_1 and B_2 will have at least one point which does not belong to any $(j-1)$-border in the graphs $G(\mathcal{B})$. Therefore, the faces F_1 and F_2 are not connected by any path in $LG(\gamma)$. □

Once we have defined the labels, these are propagated from the graph $BCG(\gamma)$ to the graph $LG(\gamma)$ and to $G(\gamma)$ and recursively to the lower-dimensional graphs which contribute to $G(\gamma)$. We stop the construction when $j = d$ and the final object is a single graph $G(U)$ with a proper labelling together with a vertical ray-shooting data structure for U.

4.3. Point-location procedure. Using the data structure $VRS(U)$, we can find for a query point p the simplex $s \in U$ immediately below p and also the landing point p' on s. We repeat the vertical query procedure on s. Eventually, we reach a vertex of $\mathcal{A}(U)$ whose label is the (unique) label of the d-dimensional cell containing p.

4.4. Time and storage analysis. The number of sets in $\mathcal{B}_j$ is $O(n^{d-j})$. For $j = 2$, which is the base case, we spend time $O(n \log n + K_\beta)$ for each set β. Thus the total cost of the base step is

$$\sum_{\beta \in \mathcal{B}_2} O(n \log n + K_\beta) = O(n^{d-1} \log n + K).$$

The graphs $G(\mathcal{B}_j)$ for $j > 2$ have total size

$$\sum_{\beta \in \mathcal{B}_j} O(n^{j-1} + K_\beta) = O(n^{d-1} + K).$$

We keep a dictionary of faces such that for each face, we can efficiently retrieve the corresponding borders and the graphs $G(\mathcal{B}_j)$ having these borders as nodes. We can use standard data structure with logarithmic cost per operation. Thus we bound the time to construct $G(\Gamma)$ with $O(n^{d-1} \log n + K \log n)$. The additional vertical ray-shooting data structures are set up at cost

$$\sum_{\beta \in \mathcal{B}_j} O(n^{j-1+\epsilon} + K_\beta) = O(n^{d-1+\epsilon} + K).$$

Labelling steps I and II are based on depth-first search [AHU74] and take overall time proportional to the size of $G(\mathcal{B})$. The ray-shooting operations in labelling step II take $O(\log^3 n)$. The number of vertical ray-shooting queries is $O(n)$ for each $\beta \in \mathcal{B}_j$ because we have always at most n components. Thus the total cost is $O(n^{d-j+1} \log^3 n) = O(n^{d-1} \log^3 n)$. The graphs $BCG(\beta)$ have only $O(n)$ nodes; therefore, finding the components takes total time $O(n^{d-j+1}) = O(n^{d-1})$ [AHU74]. The correctness of the algorithm has been established in the previous section. The analysis of the storage and preprocessing time gives us the following theorem.

THEOREM 3. *Given a set U of n simplices in E^d, let K be the number of cells of any dimension in $\mathcal{A}(U)$. For every $\epsilon > 0$, it is possible to build a point-location data structure of size $O(n^{d-1+\epsilon} + K)$ in time $O(n^{d-1+\epsilon} + K \log n)$, where the constant of proportionality depends on ϵ, such that a unique identifier for the cell containing a query point p can be found in time $O(\log^3 n)$.*

We obtain the following corollary, which exploits the point-location data structure to solve motion-planning problems.

COROLLARY 1. *Using the data structure of Theorem 3, given two points p_1 and p_2, we can determine in time $O(\log^3 n)$ whether there is a collision-free path from p_1 to p_2.*

Since we can store spanning trees of the graphs built in the several phases of the algorithms, we can find an explicit representation of a path connecting two query points. The length of the path is $O(n^{d-1} \log n)$, as follows from the bound in [AS92].

5. A method for building triangulations.

In this algorithm, by a d-cell, we indicate a truncated vertical prism whose cross-section is a $(d-1)$-simplex. We are given a set U of n pairwise-interior-disjoint $(d-1)$-simplices in E^d. The construction of the triangulation $T(U)$ of $E^d \setminus U$ proceeds in stages. We build a sequence of sets $C_0, \dots, C_l$, where $l = \log_{r_0} n$ and r_0 is a suitable constant. The basis of the construction is C_0, and we proceed inductively from C_{k-1} to build C_k. The set C_k is a collection of triples $(s, P(s), Q(s))$, where s is a d-cell, $P(s)$ is the subset of $(d-1)$-simplices in U *partially covering* s, and $Q(s)$ is the subset of $(d-1)$-simplices in U *fully covering* s. If $s \in C_k$, we will have associated sets $P(s)$ and $Q(s)$ with the invariant properties that

(I) $|P(s)| \leq n/{r_0}^k$ (first invariant),

(II) $|Q(s)| \leq nr_0/{r_0}^k$ (second invariant).

C_k is a refinement of C_{k-1} and, moreover, the d-cells in C_k *partition* E^d. At the last stage, C_l contains only d-cells intersecting a constant number of original simplices in U, and thus we obtain a triangulation $T(U)$ of size $O(|C_l|)$ by a final greedy decomposition. The first set is $C_0 = \{(E^d, U, \emptyset)\}$, which satisfies invariants (I) and (II). We assume that the two invariants hold for C_{k-1} with $k > 0$, and we show how to construct C_k.

5.1. The algorithm.

Let s be an elementary cell in C_{k-1}. Since the the first and second invariants hold by the inductive hypothesis, s is partially covered by at most $n/{r_0}^{k-1}$ $(d-1)$-simplices and is fully covered by at most $nr_0/{r_0}^{k-1}$ $(d-1)$-simplices. The covering $(d-1)$-simplices $Q(s)$ are disjoint, and therefore they are linearly ordered in the vertical direction within s. We split $Q(s)$ into at most r_0

groups of $\lceil Q(s)/r_0 \rceil$ simplices each by selecting every $\lceil Q(s)/r_0 \rceil$th $(d-1)$-simplex in the vertical ordering. The selected totally covering $(d-1)$-simplices slice s into elementary cells numbered $i = 1, \ldots, r_0$, which we denote $\sigma(s,i)$ (or σ_i whenever s is clear from the context). We also denote by $\sigma'(s,i)$ the vertical projection of $\sigma(s,i)$.

From the above construction, each σ_i is covered by $|Q(\sigma_i)| \leq nr_0/{r_0}^k$ $(d-1)$-simplices. Each σ_i is partially intersected by $|P(\sigma_i)| = q_i$ $(d-1)$-simplices, where $\sum_i q_i = |P(s)| \leq n/{r_0}^{k-1}$. This property is easily proved since a partially covering $(d-1)$-simplex of $P(s)$ can intersect to only one elementary cell σ_i.

The average σ_i has a partial cover of size $|P(s)|/r_0$, which is exactly what is needed to have a valid element of the set C_k. To make this averaging argument work in the worst case, we proceed as follows for each elementary cell σ_i. If $P(\sigma_i) < n/r_0^k$, we do nothing. Otherwise, we project independently for each cell σ_i all the $(d-2)$-boundaries of the partially covering simplices of $P(\sigma_i)$ onto a $(d-1)$-subspace. We extend these sets into full hyperplanes, obtaining a set $P'(s,i)$ of q_i hyperplanes in $(d-1)$-space. Since $n/r^k \leq |P(\sigma_i)| \leq n/r^{k-1}$, we have the conditions for using Lemma 1. We build the sparse net of Lemma 1 in $(d-1)$-space and its triangulation. Thus we generate elementary $(d-1)$-dimensional cells which are $(d-1)$-simplices. Then each such cell is extended in the vertical direction and intersected with σ_i, thus obtaining a into truncated d-prism.

We choose as a parameter of the sparse-net construction a number ρ_{0i} such that $q_i/\rho_{0i} = |P(s)|/r_0$, and thus $\rho_{0i} = q_i r_0/|P(s)|$. The interesting property we use in the analysis is that the sum of the ρ_{0i}'s is less than r_0:

$$\sum_i \rho_{0i} = \sum_i q_i r_0/|P(s)| = r_0/|P(s)| \sum_i q_i \leq r_0|P(s)|/|P(s)| = r_0.$$

Let η be a cell so obtained. We have that the number of $(d-1)$-simplices partially covering η is $|P(\eta)| \leq q_i/\rho_{0i} = P(s)/r_0 = n/{r_0}^k$ (Lemma 4); thus η satisfies the first invariant of C_k. The number of $(d-1)$-simplices in $Q(\eta)$ is at most $|Q(s)|/r_0 + |P(s)|$ and thus is at most $n/{r_0}^{k-1} + n/{r_0}^{k-1} = 2nr_0/{r_0}^k$.

We take η and use the median simplex in the vertical ordering of $Q(\eta)$ to split η into two cells η_1 and η_2 which satisfy both invariants for C_k. Collecting all of these cells we have C_k, which is a partition of E^d. This is the end of the algorithm that builds C_k starting from C_{k-1}.

5.2. Analysis of the algorithm. We now derive a recursive equation linking the size of C_k with the size of C_{k-1} by summing the number of new cells produced by the algorithm for each cell s in C_{k-1}. Let $A_i(s)$ be the number of d-dimensional cells generated within $\sigma(s,i)$ in one phase of the algorithm. Using the bound of Lemma 2, the number of cells obtained is

$$A_i(s) \leq c_9[\rho_i^{d-2} + (\rho_i/q_i)^{d-1} v(P'(s,i), \sigma'(s,i)) + \rho_i^{d-1}/\rho_{0i}],$$

where $\rho_i = \rho_{0i} \log \rho_{0i}$. Thus the total number of cells in C_k is bounded by

$$\begin{aligned} |C_k| &\leq 2 \sum_{s \in C_{k-1}} \sum_{\sigma(s,i)} A_i(s) \\ &= \sum_{s \in C_{k-1}} \sum_{\sigma(s,i)} c_{10}[\rho_i^{d-2} + (\rho_i/q_i)^{d-1} v(P'(s,i), \sigma'(s,i)) + \rho_i^{d-1}/\rho_{0i}]. \end{aligned}$$

Bounding terms of the summation separately, we have

$$\sum_i \rho_i^{d-2} \leq \left(\sum_i \rho_{0i}\log\rho_{0i}\right)^{d-2} \leq {r_0}^{d-2}\log^{d-2} r_0$$

and

$$\sum_i \rho_i^{d-1}/\rho_{i0} \leq \left(\left(\sum_i \rho_{0i}\right)^{d-2}\right)\left(\log^{d-1}\left(\sum_i \rho_{0i}\right)\right) \leq {r_0}^{d-2}\log^{d-1} r_0.$$

Since $\rho_i/q_i \leq {r_0}^k(\log r_0)/n$, we obtain a term

$$\sum_s\sum_i({r_0}^{k(d-1)}(\log r_0)^{d-1}/n^{d-1})v(P'(s,i),\sigma'(s,i)).$$

Since $\sum_s\sum_i v(P'(s,i),\sigma'(s,i)) = O(n^{d-1})$, we obtain the following recursive inequality:

$$|C_k| \leq c_{11}(\log r_0)^{d-1}{r_0}^{k(d-1)} + c_{11}(\log r_0)^{d-1}{r_0}^{d-2}|C_{k-1}|$$

for a constant c_{11} which is independent of r_0. In the next lemma, we solve the recurrence, obtaining a bound for $|C_k|$ as a function of k.

LEMMA 7. $|C_k| \leq D{r_0}^{k(d-1)}$ *for a constant* D *independent of* k.

Proof. We use an induction on k. For $k = 0$, $C_0 = (E^d, U, \emptyset)$, so $|C_0| \leq D$ for $D \geq 1$. Inductively, we assume the bound on $|C_{k-1}|$.

$$\begin{aligned}|C_k| &\leq c_{11}(\log r_0)^{d-1}{r_0}^{k(d-1)} + c_{11}D(\log r_0)^{d-1}{r_0}^{d-2}{r_0}^{(k-1)(d-1)}\\ &\leq c_{11}(\log r_0)^{d-1}{r_0}^{k(d-1)} + (c_{11}D(\log r_0)^{d-1}/r_0){r_0}^{k(d-1)}.\end{aligned}$$

Choosing r_0 and D large enough, we can make sure that

$$c_{11}(\log r_0)^{d-1} + c_{11}D(\log r_0)^{d-1}/r_0 \leq D,$$

and thus the bound is proved. □

The total number of cells in the sequence of sets C_k is a summation of a geometric sequence of ratio $r_0^{d-1} > 1$, so its value is proportional to the last term, which is $O(n^{d-1})$.

$$\sum_{k=0}^{l}|C_k| \leq \sum_{k=0}^{l} Dr_0^{k(d-1)} \leq 2D{r_0}^{l(d-1)} = O(n^{d-1}).$$

What is the time to compute all of the C_i's? The sparse nets are computed in time linear in the number of simplices partially covering the elementary cells by Lemma 3. The slicing hyperplanes are found by repeated applications of a selection algorithm in time $O(r_0|Q(s)|)$ for any elementary cell s. Thus we have running time

$$\sum_{k=0}^{l}((n/{r_0}^k) + (n{r_0}^2/{r_0}^k))|C_k| \leq 2(1+{r_0}^2)Dn{r_0}^{l(d-2)} = O(nn^{d-2}) = O(n^{d-1}).$$

The above discussion proves the following theorem.

THEOREM 4. *Give a set U of n pairwise-interior-disjoint $(d-1)$-simplices in d-space with $d \geq 3$, we can deterministically build a triangulation $T(U)$ of size $O(n^{d-1})$ in time $O(n^{d-1})$.*

Following the tree structure of the triangulation, we can locate the cells in $T(U)$ containing a query point p in time $O(\log n)$.

COROLLARY 2. *We can find in time $O(\log n)$ the cells of $T(U)$ containing a query point p and the simplex in U immediately below p.*

6. Conclusions. In this paper, we have presented algorithms for solving point-location and motion-planning queries in arrangements of simplices in E^d for any fixed $d \geq 2$. The preprocessing time and storage bounds depend explicitly on the complexity of the arrangement induced by the simplices. This feature is important in order to exploit the "sparsity" of arrangements which are obtained, for example, in motion-planning and visibility applications. For sets of interior-disjoint simplices in E^d with $d \geq 3$, we improve bounds for building triangulations, and we obtain a fast point-location method as well.

Acknowledgments. I wish to thank Bruno Codenotti and Nina Amenta for their observations on an early draft and the referees for their useful comments.

REFERENCES

[AHU74] A. V. AHO, J. E. HOPCROFT, AND J. D. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison–Wesley, Reading, MA, 1974.

[Ale56] P. S. ALEXANDROV, *Combinatorial Topology*, Graylock Press, Rochester, NY, 1956.

[AM92] P. K. AGARWAL AND J. MATOUŠEK, *Ray shooting and parametric search*, in Proc. 24th Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1992, pp. 517–526.

[AS90] B. ARONOV AND M. SHARIR, *Triangles in space or building (and analyzing) castles in the air*, Combinatorica, 10 (1990), pp. 137–173

[AS92] ———, *Castles in the air revisited*, in Proc. 8th ACM Symposium on Computational Geometry, Association for Computing Machinery, New York, 1992, pp. 146–256.

[Ber93] M. BERN, *Compatible tetrahedralizations*, in Proc. 9th ACM Symposium on Computational Geometry, Association for Computing Machinery, New York, 1993, pp. 281–288.

[Can87] J. CANNY, *The Complexity of Robot Motion Planning*, MIT Press, Cambridge, MA, 1987.

[CE92] B. CHAZELLE AND H. EDELSBRUNNER, *An optimal algorithm for intersecting line segments in the plane*, J. Assoc. Comput. Mach., 39 (1992), pp. 1–54.

[CEGS89] B. CHAZELLE, H. EDELSBRUNNER, L. GUIBAS, AND M. SHARIR, *Algorithms for bichromatic line segment problems and polyhedral terrains*, Report UIUCDCS-R-90-1578, Department of Computer Science, University Illinois at Urbana–Champaign, Urbana, IL, 1989.

[CF92] B. CHAZELLE AND J. FRIEDMAN, *Point location among hyperplanes and vertical ray-shooting*, Comput. Geom., in press.

[Cha84] B. CHAZELLE, *Convex partitions of polyhedra: A lower bound and worst-case optimal algorithm*, SIAM J. Comput., 13 (1984), pp. 488–507.

[Cha87] ———, *Approximation and decomposition of shapes*, in Advances in Robotics, Vol. 1: Algorithmic and Geometric Aspects of Robotics, Lawrence Erlbaum Associates, Hillsdale, NJ, 1987, pp. 145–185.

[Cha93] ———, *Cutting hyperplanes for divide and conquer*, Discrete Comput. Geom., 9 (1993), pp. 145–158.

[Cla87] K. L. CLARKSON, *New applications of random sampling in computational geometry*, Discrete Comput. Geom., 2 (1987), pp. 195–222.

[Col86] R. COLE, *Searching and storing similar lists*, J. Algorithms, 7 (1986), pp. 202–220.

[dBGH94] M. DE BERG, L. GUIBAS, AND D. HALPERIN, *Vertical decompositions for triangles in 3-space*, in Proc. 10th ACM Symposium on Computational Geometry, Association for Computing Machinery, New York, 1994.

[Ede87] H. Edelsbrunner, *Algorithms in Combinatorial Geometry*, Springer-Verlag, Berlin, New York, Heidelberg, 1987.

[EGS86] H. Edelsbrunner, L. Guibas, and J. Stolfi, *Optimal point location in a monotone subdivision*, SIAM J. Comput., 15 (1986), pp. 317–339.

[GT91] M. Goodrich and R. Tamassia, *Dynamic trees and dynamic point location*, in Proc. 23rd Annual ACM Symposium on Theory of Computing, 1991, pp. 523–533.

[Kir83] D. Kirkpatrick, *Optimal search in planar subdivision*, SIAM J. Comput., 12 (1983), pp. 28–35.

[Lat91] J.-C. Latombe, *Robot Motion Planning*, Kluwer Academic Publishers, Norwell, MA, 1991.

[LP77] D. Lee and F. Preparata, *Location of a point in a planar subdivision and its applications*, SIAM J. Comput., 6 (1977), pp. 594–606.

[LT80] R. Lipton and R. E. Tarjan, *Applications of a planar separator theorem*, SIAM J. Comput., 9 (1980), pp. 614–627.

[Mat91] J. Matoušek, *Cutting hyperplane arrangements*, Discrete Comput. Geom., 6 (1991), pp. 385–406.

[Mat92] ———, *Range searching with efficient hierarchical cuttings*, in Proc. 8th ACM Symposium on Computational Geometry, Association for Computing Machinery, New York, 1992, pp. 276–285.

[Mat93] ———, *On vertical ray shooting in arrangements*, Comput. Geom., 2 (1993), pp. 279–285.

[Meh84] K. Mehlhorn, *Multidimensional Searching and Computational Geometry*, Springer-Verlag, Berlin, New York, Heidelberg, 1984.

[Mul91] K. Mulmuley, *Hidden surface removal with respect to a moving view point*, in Proc. 23rd Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1991, pp. 512–522.

[Pre81] F. Preparata, *A new approach to planar point location*, SIAM J. Comput., 10 (1981), pp. 473–482.

[PS85] F. P. Preparata and M. I. Shamos, *Computational Geometry: An Introduction*, Springer-Verlag, Berlin, New York, Heidelberg, 1985.

[PT88] F. Preparata and R. Tamassia, *Fully dynamic techniques for point location and transitive closure in planar structures*, in Proc. 29th IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1988, pp. 558–567.

[PT89] ———, *Efficient spatial point location*, in Proc. 1989 Workshop on Algorithms and Data Structures, Lecture Notes in Comput. Sci. 382, Springer-Verlag, Berlin, 1989, pp. 3–11.

[PY90] M. S. Paterson and F. F. Yao, *Efficient binary space partitions for hidden surface removal and solid modeling*, Discrete Comput. Geom., 5 (1990), pp. 485–503.

[Sch92] O. Schwarzkopf, *Ray-shooting in convex polytopes*, in Proc. 8th ACM Symposium on Computational Geometry, Association for Computing Machinery, New York, 1992, pp. 286–295.

[SS83] J. T. Schwartz and M. Sharir, *On the piano mover's problem II General techniques for computing topological properties of real algebraic manifolds*, Adv. Appl. Math., 4 (1983), pp. 298–351.

[SS89] ———, *A survey of motion planning and related geometric algorithms*, in Geometric Reasoning, D. Kapur and J. L. Mundy, eds., MIT Press, Cambridge, MA, 1989, pp. 157–169.

[SS90] ———, *Algorithmic motion planning in robotics*, in Handbook of Theoretical Computer Science, vol. A, J. van Leeuwen, ed., Elsevier, Amsterdam, 1990, pp. 157–169.

[ST86] N. Sarnak and R.E. Tarjan, *Planar point location using persistent search trees*, Comm. Assoc. Comput. Mach., 29 (1986), pp. 669–679.

SIAM J. COMPUT.
Vol. 25, No. 5, pp. 1082–1104, October 1996

008

BOUNDS ON THE EFFICIENCY OF MESSAGE-PASSING PROTOCOLS FOR PARALLEL COMPUTERS*

ROBERT CYPHER† AND SMARAGDA KONSTANTINIDOU†

Abstract. This paper considers the problem of creating message-passing protocols for parallel computers. It is assumed that the processors are connected by a network that provides guaranteed delivery of every message, provided that each message delivered by the network is removed by the receiving processor unconditionally and in finite time. Two models of message passing are considered, namely, a selective model in which the receiver specifies the source of the message and a nonselective model in which the receiver accepts messages from all sources. We consider only space-efficient protocols in which each processor has storage for a constant number of messages and message headers. We present three main results. First, we give a protocol for the selective model that performs a constant amount of communication per send or receive posted by the application. Second, we prove that no such efficient protocol exists for the nonselective model. Third, we present a protocol for the nonselective model that performs a logarithmic amount of communication per send or receive posted by the application.

Key words. communication protocols, end-to-end protocols, message passing, parallel computing deadlock

AMS subject classifications. 68Q10, 68Q22, 68Q25

1. Introduction.

A standard model for communication in both parallel and distributed computers is the message-passing model, in which processes communicate solely by posting matching send and receive commands [2]. The implementation of a message-passing model requires a *communication protocol* that guarantees the delivery of messages between matching send and receive commands.

Numerous communication protocols have been proposed and their properties studied, particularly in the context of communication networks [13, 16]. A significant amount of work has also been done in the area of communication protocols for loosely coupled multiprocessors interconnected via a local-area network or token ring [1, 15, 18, 19]. In the area of message-passing parallel computers, the assumptions regarding the properties of the system can be substantially different from those used in the area of distributed systems [5, 11]. In particular, the interconnection networks of most parallel machines in existence use blocking routing algorithms in which messages that encounter congestion are not discarded by the network. Instead, these messages continue to hold resources until they can be serviced. As a result, the interconnection network is typically guaranteed to be free of deadlock only if the following *continuous-consumption* requirement is met: every message delivered by the network is removed by the receiving processor *unconditionally* and in finite time [8]. Furthermore, it is common for the processors in a parallel machine to have local memory but no disks. As a result, the communication protocol must make efficient use of the limited memory that is available. This paper considers the problem of creating message-passing protocols for parallel computers that are memory efficient and that satisfy the continuous-consumption requirement.

One simple type of communication protocol for n processors requires the static allocation of n storage *buckets* per processor, where each bucket can store one or more

* Received by the editors January 26, 1994; accepted for publication (in revised form) January 25, 1995. A preliminary version of this paper appeared in *Proc.* 1993 *ACM Symposium on Parallel Algorithms and Architectures.*

† Department of Computer Science, Johns Hopkins University, 3400 North Charles Street, Baltimore, MD 21218 (cypher@cs.jhu.edu, konstant@cs.jhu.edu)

messages [16]. The storage in these buckets can be managed with a *sliding-window* protocol in which the receiving processor grants tokens for additional messages to each sending processor. Such protocols have been used in parallel computers [14], but the requirement of n storage buckets per processor limits their scalability. Although it is relatively easy to obtain a protocol that requires only n bits per processor (rather than n buckets), it is natural to ask if protocols exist that have constant storage requirements per processor.

In fact, a protocol requiring constant storage per processor was developed in the context of distributed operating systems [3]. In this protocol, each processor maintains a list of the messages that have been sent to the processor but have not yet been received. This list is distributed among the processors sending messages to the given processor in order to reduce the memory requirements. Specifically, if message M_1 originating at processor S_1 arrives at the destination processor R before R posts a receive, M_1 is discarded and a record of the request is kept in R. If another processor S_2 sends a message M_2 to R before R has consumed M_1, M_2 is also discarded and a control message is sent from R to S_1 with the address of S_2. If yet another processor S_3 sends a message M_3 to R before R has consumed M_1, M_3 is also discarded and a control message is sent from R to S_2 with the address of S_3. Then, when R retrieves message M_1 from S_1, it also receives the address of S_2, which is the next processor in the distributed list of outstanding messages for R. Similarly, when R retrieves the message M_2 from S_2, it also receives the address of S_3. In general, R maintains a linked list of processors attempting to send it messages by storing the addresses of only the first and last processors in the list.

Unfortunately, this protocol does not satisfy the continuous-consumption requirement stated above. In particular, R must send out a message for each message that arrives at R after M_1 arrives. Because R must generate a message for each additional message that it receives, it can receive messages only as fast as it can send new messages into the network. As a result, this protocol could deadlock if it were implemented on top of a blocking point-to-point or multistage communication network, such as the ones typically used by current multicomputers.

In addition, other researchers have addressed the deadlock problems that are caused by blocking networks and finite storage at the sending and receiving processors [7, 8, 10]. However, these papers do not consider the implementation of message-passing primitives.

The remainder of this paper is organized as follows. In §2, formal definitions of the model and of the problems being addressed are presented. An efficient protocol for selective receives is given in §3. A proof that no such efficient protocol is possible for nonselective receives is given in §4. Finally, a protocol that requires a logarithmic number of messages per send or receive posted by the application is presented in §5.

2. Formal model and problem definitions.

2.1. The model.

We will consider the implementation of a single parallel application. The application consists of n processes that operate asynchronously and communicate with one another solely by issuing send and receive commands. Associated with each application process is a communication process that implements the communication protocol. An application process communicates by posting a send or receive command to its associated communication process and then blocking until it is unblocked by its communication process.

The n communication processes communicate with one another by sending and receiving messages over an interconnection network. Each communication process has a single outgoing port for sending messages to the network and a single incoming port for receiving messages from the network. Each of these ports has a strictly first-in first-out (FIFO) queue that can hold a constant number of messages. A communication process sends a message to another process by placing the body of the message along with a header giving the address of the destination process into its outgoing FIFO queue (provided that it is not full).

We will assume that every application message is of the same size and that each communication process has storage for a constant number of application messages and message headers. Application messages and message headers are assumed to be indivisible units that cannot be merged or encoded in any way. More specifically, the only operations that a communication process can apply to an application message or a message header are store, copy, discard, send, and receive. The body of a message sent through the interconnection network can contain a constant number of application messages and message headers.

The interconnection network provides full connectivity between communication processes and is guaranteed to deliver every message within finite time, provided that the communication processes guarantee continuous consumption of messages. In addition, the network is *nonovertaking.* That is, multiple messages sent from the same source to the same destination arrive in the order in which they were sent.

2.2. The problems. We will consider two message-passing models. The first model consists of the commands SEND(M, R) and SRECV(S). The command SEND(M, R) sends the application message M to process R. The command SRECV(S), called a "selective receive," receives an application message from process S. Both the SEND and SRECV commands block the application process that posts them until a matching SRECV or SEND command has been posted. It will be assumed that for each SEND (respectively, SRECV) that the application posts, a matching SRECV (respectively, SEND) will eventually be posted.

The second model consists of the commands SEND(M, R) and NRECV(). The command SEND(M, R) sends the application message M to process R. The command NRECV(), called a "nonselective receive," receives an application message from any process. Both the SEND and NRECV commands block the application process that posts them until a matching NRECV or SEND command has been posted. It will be assumed that for each SEND (respectively, NRECV) that the application posts, a matching NRECV (respectively, SEND) will eventually be posted.

For both models, an application process may post either a finite number or an infinite number of send and receive commands. The communication protocol is required to guarantee that for every SEND(M, R) command posted, the message M is transferred within finite time to the process R that issues the matching receive command. In addition, the communication protocol must unblock the application process within finite time after its posted send or receive has been completed.

3. Upper bound for selective receives.

In this section, we present a protocol for SEND and SRECV that requires only constant storage per process and that performs only a constant amount of communication per SEND or SRECV posted by the application.

The protocol uses four types of control messages, called READY, REQ1, REQ2, and ACK, and two types of data messages, called DATA1 and DATA2. The protocol operates as a finite-state machine where the number of states (eight) is independent of the number of processes. Initially, all of the communication processes are in state

C_0. The protocol performs the following operations based on the state of the communication process:

State C_0. Discard all arriving messages. If the associated application process posts a SEND, enter state S_0. If the associated application process posts a SRECV, enter state R_0.

When the application process has a posted SEND, the communication process is in one of the following states. Let R denote the destination of the posted SEND and let M denote the message being sent.

State S_0. Send a READY message to R. Discard all arriving messages that are not from R. If a REQ1 message from R arrives, enter state S_1. If a REQ2 message from R arrives, enter state S_2.

State S_1. Send a DATA1 message containing M to R. Discard all arriving messages that are not from R. If a REQ2 message from R arrives, enter state S_2. If an ACK message from R arrives, enter state S_3.

State S_2. Send a DATA2 message containing M to R. Discard all arriving messages except an ACK from R. When an ACK message from R arrives, enter state S_3.

State S_3. Unblock the associated application process. Enter state C_0.

When the application process has a posted SRECV, the communication process is in one of the following states. Let S denote the source specified by the posted SRECV.

State R_0. Send a REQ1 message to S. Discard all arriving messages that are not from S. If a READY message from S arrives, enter state R_1. If a DATA1 message from S arrives, enter state R_2.

State R_1. Send a REQ2 message to S. Discard all arriving messages except a DATA2 message from S. When a DATA2 message from S arrives, enter state R_2.

State R_2. Send an ACK message to S. Copy the data from the most recent (DATA1 or DATA2) message to the application. Unblock the associated application process. Enter state C_0.

Three possible executions of this protocol are illustrated in Figures 1, 2, and 3. These figures, in which time moves downward, show the actions and states of two processes that execute matching SEND and SRECV commands. Messages from other processes are discarded and are not shown.

The idea behind this protocol is that both the sender and the receiver initiate the communication. If the READY message from the sender arrives at the receiver before the matching SRECV is posted, the receiver discards the READY message (because the receiver does not yet know that it will want to communicate with the given sender). However, in this case, the SRECV will eventually be posted, at which point the receiver sends a REQ1 message to the sender and the communication will be established (see Figure 1). The situation in which the REQ1 message from the receiver arrives before the matching SEND is posted is analogous (see Figure 2). The need for two types of data messages (DATA1 and DATA2) arises because of the possibility of the SEND and matching SRECV being posted nearly simultaneously (see Figure 3). In particular, note that in both Figures 2 and 3, the receiver posts an SRECV, sends a REQ1 message, receives a READY message, sends a REQ2 message, and then receives a data message. If only one type of data message were used, the receiver would not be able to distinguish between these two cases and would not know if it should unblock its application or wait for another data message from the sender.

We will now prove that the above protocol is correct. We will assume for Theo-

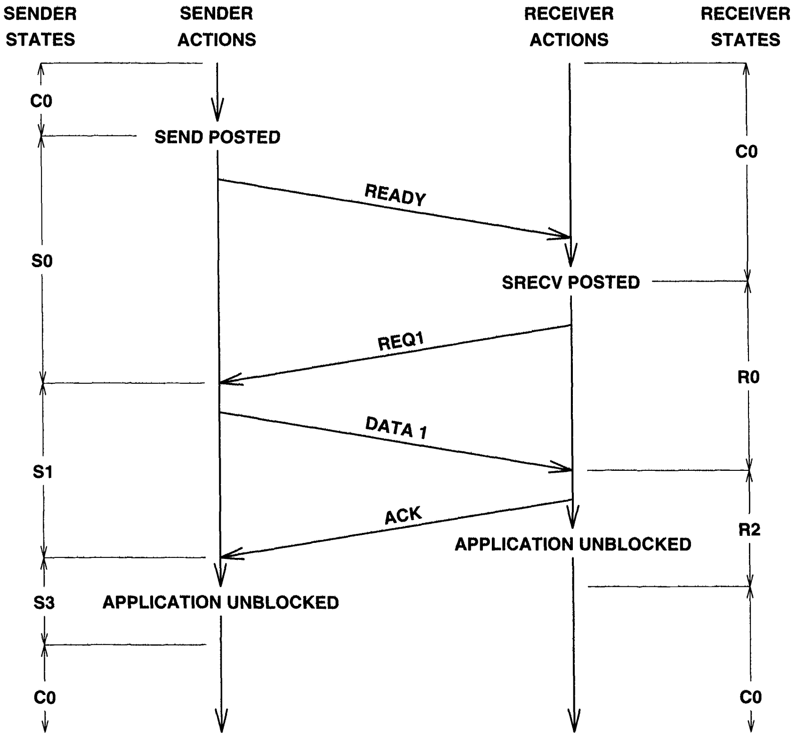


FIG. 1. *A possible execution of the selective receive protocol in which the* READY *message arrives before the* SRECV *is posted.*

rem 3.1 that each communication process has sufficient storage in its outgoing FIFO queue for all of the messages that it sends. We will bound the storage and communication requirements in Theorem 3.2.

THEOREM 3.1. *The above protocol is a correct implementation of* SEND *and* SRECV. *Furthermore, once an application process becomes unblocked after posting a* SEND (SRECV), *its communication process will not receive any messages that were generated by the matching* SRECV (SEND).

Proof. The proof is by contradiction. If the theorem does not hold, there must exist a process S that posts a SEND of a message M to a process R, where R posts a matching SRECV from S and where the theorem fails for this SEND–SRECV pair but does hold for all previous SEND–SRECV pairs between these two processes. Consider such a SEND–SRECV pair, and note that once S posts this SEND, the first message that it will receive from R will be generated by the matching SRECV. Similarly, once R posts this SRECV, the first message that it will receive from S will be generated by the matching SEND.

We will now consider how the theorem could fail to hold for this SEND–SRECV pair. When S posts this SEND to R, S enters state S_0 and sends a READY message to R. There are two cases based on when this READY message arrives at R.

Case 1. In this case, the READY message from S arrives at R before R posts its matching SRECV from S (see Figure 1). This implies that the READY message will be discarded by R, R will post its matching SRECV from S, R will enter state R_0, and R will send a REQ1 message to S. This implies that the REQ1 message will

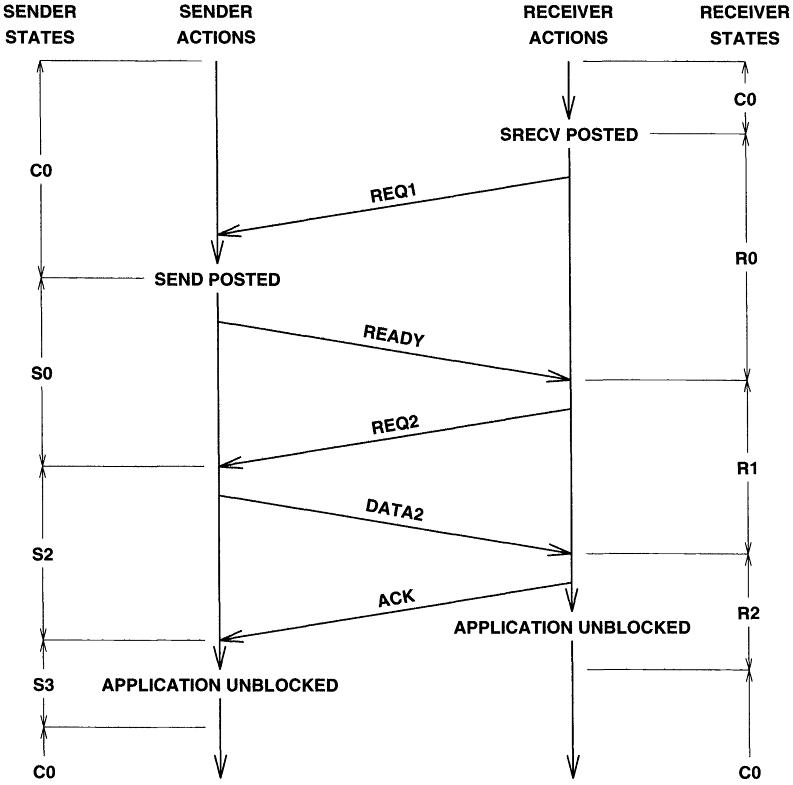


FIG. 2. *A possible execution of the selective receive protocol in which the* REQ1 *message arrives before the* SEND *is posted.*

arrive at S when S is in state S_0, S will enter state S_1, and S will send a DATA1 message containing M to R. This implies that the DATA1 message will arrive at R when R is in state R_0, R will enter state R_2, R will send an ACK message to S, R will copy M to its application process, R will unblock its application process, and R will enter state C_0. This implies that the ACK message will arrive at S when S is in state S_1, S will enter state S_3, S will unblock its application process, and S will enter state C_0. Note that in this case, S receives and consumes all messages issued by R as a consequence of the SRECV (namely REQ1 and ACK) before it unblocks its application process. Similarly, R receives and consumes all messages issued by S as a consequence of the SEND (namely READY and DATA1) before it unblocks its application process.

Case 2. In this case, the READY message arrives at R after R has posted the matching SRECV from S, sent a REQ1 message to S, and entered state R_0. Then there are two subcases:

Case 2a. The REQ1 message from R arrived at S before S posted the matching SEND to R (see Figure 2). This implies that the REQ1 message was discarded by S. Then the READY message arrives at R when R is in state R_0, R sends a REQ2 message to S, and R enters state R_1. The REQ2 message finds S in state S_0 and results in S entering state S_2 and sending a DATA2 message to R. This implies that the DATA2 message will arrive at R when R is in state R_1, R will enter state R_2, R will send an ACK message to S, R will copy M to its

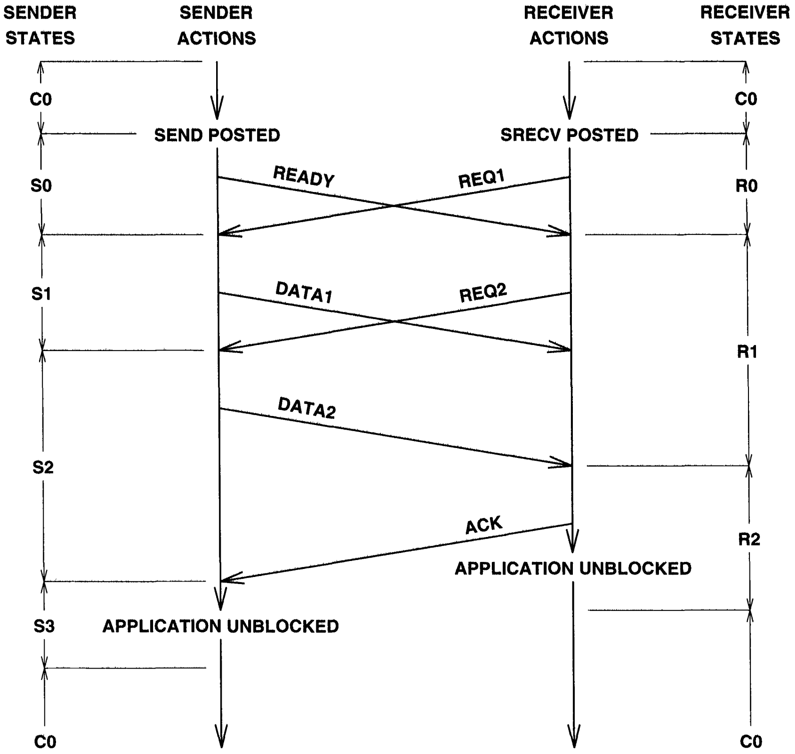


FIG. 3. *A possible execution of the selective receive protocol in which the* SEND *is posted before the* REQ1 *message arrives and the* SRECV *is posted before the* READY *message arrives.*

application process, R will unblock its application process, and R will enter state C_0. This implies that the ACK message will arrive at S when S is in state S_2, S will enter state S_3, S will unblock its application process, and S will enter state C_0. Note that in this case, S receives and consumes all messages issued by R as a consequence of the SRECV (namely REQ1, REQ2, and ACK) before it unblocks its application process. Similarly, R receives and consumes all messages issued by S as a consequence of the SEND (namely READY and DATA2) before it unblocks its application process.

Case 2b. In this subcase, the REQ1 message from R arrived at S after S posted the matching SEND to R and thus finds S in state S_0 (see Figure 3). Then, possibly simultaneously, S enters state S_1 and sends a DATA1 message to R and R enters state R_1 and sends a REQ2 message to S. Upon the arrival of the REQ2 message, S enters state S_2 and sends a DATA2 message to R. Possibly simultaneously, R receives the DATA1 message while in state R_1 and discards it. When the DATA2 message arrives at R, R is still in state R_1. Then it enters state R_2, it sends an ACK message to S, R copies M to its application process, R unblocks its application process, and R enters state C_0. This implies that the ACK message will arrive at S when S is in state S_2, S will enter state S_3, S will unblock its application process, and S will enter state C_0. Note that in this case, S receives and consumes all messages issued by R as a consequence of the SRECV (namely REQ1, REQ2, and ACK) before it unblocks its application process. Similarly, R receives and consumes all messages issued by S as a consequence of the SEND (namely READY, DATA1, and DATA2) before it unblocks its application process.

Note that in every case, the following hold:

1. The SEND (SRECV) blocks until the matching SRECV (SEND) is posted.
2. The SEND (SRECV) is unblocked within finite time after the matching SRECV (SEND) is posted.
3. The message sent by the SEND is transferred to the matching SRECV.
4. Once an application process becomes unblocked after posting a SEND (SRECV), its communication process will not receive any messages that were generated by the matching SRECV (SEND).

Therefore, the theorem holds for this SEND–SRECV pair, which is a contradiction. □

THEOREM 3.2. *The above protocol requires only constant storage per process and performs only a constant amount of communication per* SEND *or* SRECV *posted by the application.*

Proof. The protocol consists of eight states and each state consists of at most four steps. Therefore, the protocol requires only a constant amount of control storage per process. By examining the possible interactions between S and R given in the proof of Theorem 3.1, it is clear that a communication process will send at most three messages in response to the posting of a SEND or SRECV. Therefore, the protocol performs only a constant amount of communication per SEND or SRECV posted by the application.

Furthermore, it is also clear from examining the possible interactions between S and R given in the proof of Theorem 3.1 that a communication process A which is acting in response to a SEND (SRECV) does not unblock its application process until the process B that is executing the matching SRECV (SEND) has received at least one message from A generated by the SEND (SRECV) currently posted at A. Therefore, by the time a communication process unblocks its application process for the ith time, all of the messages sent by the communication process in response to the first $i-1$ postings of SENDs and SRECVs at that process have been accepted by the network (because at least one message sent in response to the ith posting of a SEND or SRECV has arrived at the process posting the matching SRECV or SEND). More precisely, when a communication process unblocks its application process for the ith time, the communication process has sent at most two messages that have not yet been accepted by the network (because all messages from the first $i-1$ postings of SENDs and SRECVs have been accepted by the network, and at least one message from the ith posting of a SEND or SRECV has been accepted by the network). As a result, storage for at most five messages that have been sent but not yet accepted by the network (three for the $(i+1)$st SEND or SRECV and two for the ith SEND or SRECV) is sufficient to hold all of the messages generated by the $(i+1)$st SEND or SRECV). Because i was chosen arbitrarily, it follows that storage for at most five messages per process is sufficient to prevent deadlock. □

4. Lower bound for nonselective receives.

In this section, we prove that any protocol for SEND and NRECV that uses only constant storage per process cannot perform a constant amount of communication per SEND or NRECV posted by the application. For the purposes of the proof, we will focus on a simple class of applications that require a number of processes to each send a single message to process 1. The proof is by contradiction and consists of two halves. In the first half (Lemma 4.2), we assume the existence of a scalable protocol that implements such applications correctly and we prove that there is at least one such application for which the protocol is poorly suited. In the second half (Theorem 4.3), we show that given this particular application, the protocol cannot implement the application correctly, thus yielding a contradiction.

More precisely, throughout the remainder of this section, we will consider only applications in which process 0 posts one NRECV, process 1 posts one SEND to process 0 followed by x NRECVs (for some integer x), and some set of x other processes each post one SEND to process 1. Recall that in order to send a message, a communication process must provide a message header specifying the destination of the message. A central theme in our proof will be keeping track of the message headers that each communication process holds (and thus the set of processes to which it can send messages). We will call the message headers that are present in a communication process at initialization of the protocol *static headers.* Recall that a communication process can obtain new headers only by receiving them in messages. Message headers that are acquired from messages sent from other processes will be called *dynamic headers.* Also, recall that each communication process has storage for only a constant number of messages and (static and dynamic) message headers. In fact, our lower bound will apply to any protocol with these bounds on storage for messages and message headers, regardless of the amount of control storage that is available.

Let the constant c_1 be the maximum number of static headers per process, and let the constant c_2 be such that each process can store at most c_2 (static and dynamic) headers and at most c_2 data messages. Let n denote the number of processes other than process 0 and let G be the n-node directed graph such that there is an edge from node a to node b iff process a has a static header with b's address.[1] For ease of presentation, we will often refer to a process and to its corresponding node in G interchangeably. Note that G has at most $c_1 n$ edges and outdegree at most c_1.

Assume for the sake of contradiction that a protocol with constant storage per process for messages and message headers exists such that it performs a constant amount of communication per SEND or NRECV posted by the application. Note that given such an efficient protocol, if a single SEND to process 1 is posted at some process a and if no other SENDs or NRECVs have been posted, the protocol will send only a constant number of messages involving only a constant number of processes. We will call a's *region* the set of processes that are involved sending and/or receiving at least one of the messages caused by such a SEND being posted at process a. (If the protocol depends on the initial state of the processes and the communication network and/or if it depends on random bits, select an arbitrary initial state and set of values for the random bits and define each node's region given this initial state and these random bits.) Let the constant c_3 be such that every process has at most c_3 processes in its region. For each process a, let the *depth* of a be the number of processes b (out of the set of n possible processes) such that a is in b's region. Note that each process contributes to the depth of at most c_3 other processes (namely, those processes in its region), so the sum of the depths of all the processes is at most $c_3 n$.

The intuition behind our lower bound is as follows. Let S be the set of x nodes with a SEND posted to node 1. Consider any node $a \in S$. Node a is unaware of whether or not other nodes have SENDs posted to node 1. As a result, node a must execute a protocol that operates correctly when it is the only node with a SEND posted to node 1. Similarly, each of the other $x - 1$ nodes in S must execute a protocol that operates correctly if it is the only node with a SEND posted to node 1.

[1] Process 0 will be ignored for the remainder of the discussion. It was introduced solely in order to force process 1 to send a message, which in turn forces process 1 to remove messages from the network in order to satisfy the continuous-consumption requirement.

We will divide the n nodes into two classes: those in a set C (which includes node 1) and those not in C. We will define C so that it includes every node that lies in the the region of two or more nodes in S.

We will consider a possible time T in the execution of the protocol at which all of the nodes outside of C were able to send all of the messages that they attempted to send but the nodes in C have not yet been able to send any messages. At time T, each node outside of C that is in the region of some node $a \in S$ has no way of knowing that a was not the only node with a SEND posted to node 1. As a result, it will only perform the communication that it would have performed if node a were the only sender. On the other hand, nodes within C may have received messages initiated by multiple nodes in S, and as a result the nodes in C may be able to detect that there is more than one sender. However, we will force the set C to be small (it will have $o(x)$ nodes), so the continuous-consumption requirement will force the nodes in C to discard most of the data messages and message headers that they receive (due to the constant storage per node). As a result, the only nodes that will still hold the application message from a certain node $a \in S$ will be in a's region and outside of C. However, the only nodes that realize that a's application message has not yet been successfully delivered to node 1 are within C. By suitably bounding the distances in the graph G, we will be able to show that any attempt by a node in C to inform a node outside of C that it must resend the application message that it holds will require too many ($\omega(x)$) messages. Lemma 4.2 and Theorem 4.3 formalize this argument. Theorem 4.1 will be useful in proving Lemma 4.2.

The following theorem follows from Turán's theorem in extremal graph theory [4].

THEOREM 4.1. *For any positive integers n and k, where $2 \leq k \leq n/16$, every graph G with n nodes and $(n^2 - n)/2 - n^2/16k$ edges contains a clique of $k + 1$ nodes as a subgraph.*

LEMMA 4.2. *Given a protocol with constant storage per process for messages and message headers that performs a constant amount of communication per* SEND *or* NRECV *posted by the application and given the definitions presented above, there exists an integer x, a set S of x nodes in G, and a set C of $o(x)$ nodes in G such that the following two properties hold:*

Property 1. *The distance in G from any node c in C to any node b not in C, where b is in the region of some node a in S, is $\omega(x)$.*

Property 2. *The distance in G from any node a not in C, where a is in the region of some node b in S, to any node c not in C, where c is in the region of some node $d \neq b$ in S, is $\omega(x)$.*

Proof. We will first have to decide on the correct value of x. In order to find x, consider the infinite sequence $y_0, y_1, y_2, \ldots$, where $y_0 = \log^* n$ and for each $i \geq 0$, $y_{i+1} = 128c_3y_i^2c_1^{y_i^4}$. Let k be the largest integer such that $y_k \leq n$ and note that $k \geq (\log^* n)/4$ (for all sufficiently large n). For each i, where $0 \leq i < k$, let R_i denote the set of all nodes with depths greater than or equal to n/y_{i+1} and less than n/y_i, and let d_i denote the sum over all nodes a in R_i of the depth of a. Because the sets R_i and R_j are disjoint if $i \neq j$, $\sum_{i=0}^{k-1} d_i \leq c_3n$. Therefore, the average value of the d_i's is at most $4c_3n/\log^* n$, which implies that there must exist an i, $0 \leq i < k$, such that $d_i \leq 4c_3n/\log^* n$. Let $y = y_i$ for such a value of i, and note that there are at most $4c_3n/\log^* n$ nodes a such that a's region contains at least one node with depth greater than or equal to $n/128c_3y^2c_1^{y^4}$ and less than n/y. Let $x = y^2$. Note that $y_k \leq n$ and $i < k$, so $128c_3y^2c_1^{y^4} \leq n$. As a result, $\log^* n \leq y \leq (\log_{c_1} n)^{1/4}$ and $(\log^* n)^2 \leq x \leq (\log_{c_1} n)^{1/2}$.

Let the set C consist of node 1 plus those nodes with depths greater than or equal to n/y. Note that the total over all nodes a in C of the depth of a is at most c_3n. Therefore, C contains at most $1 + c_3n/(n/y) = 1 + c_3y = o(x)$ nodes. For each node a, let a's *outer region* consist of the nodes in a's region that are not in C. Also, for each node a, let a's *extended region* consist of each node c such that there exists a node b in a's outer region where the distance in G from b to c is at most $x^2 = y^4$.

We now want to create the set S of x nodes that is required. First, start with all n nodes and remove from consideration each node a such that a's outer region contains at least one node with depth greater than or equal to $n/128c_3y^2c_1^{y^4}$. Note that from the definition of y, this eliminates at most $4c_3n/\log^* n = o(n)$ nodes from consideration.

Next, we will eliminate from consideration those nodes that violate Property 1. More specifically, we will remove from consideration each node a such that there exists a node b in a's outer region and a node c in C where the distance in G from c to b is $x^2 = y^4$ or less. Because C contains fewer than $2c_3y$ nodes and for each node in C there are at most $2c_1^{y^4}$ nodes that are at distance at most y^4 from the given node in C, at most $4c_3yc_1^{y^4}$ nodes are at distance at most y^4 from a node in C. Each of these nodes is in the outer region of at most $n/128c_3y^2c_1^{y^4}$ nodes that are still under consideration, so this eliminates at most $4c_3yc_1^{y^4}n/128c_3y^2c_1^{y^4} = n/32y = o(n)$ nodes from consideration. At this point, for each node a still under consideration, the distance in G from any node $c \in C$ to any node in a's outer region is at least $x^2 = \omega(x)$, so node a satisfies Property 1.

Finally, we want to select a set of x nodes that also satisfy Property 2. Note that Property 2 requires the set of x nodes in S to have outer regions that are sufficiently far from one another. In order to select such a set of x nodes, we will create a graph H that captures, for each pair of possible nodes, whether or not their outer regions are too close to one another. More precisely, create the undirected graph H where the nodes correspond to the $n - o(n)$ nodes in G that remain under consideration. For each pair of nodes a and b in H, add an edge (a, b) to H if a's extended region intersects b's outer region. Note that a's extended region contains at most $2c_3c_1^{y^4}$ nodes, and each of these nodes is in the outer region of at most $n/128c_3y^2c_1^{y^4}$ nodes in H, so examining a's extended region adds at most $2c_3c_1^{y^4}n/128c_3y^2c_1^{y^4} = n/64y^2$ edges to H. Therefore, examining all $n - o(n)$ extended regions of nodes in H adds at most $n^2/64y^2$ edges to H. Let $H' = \overline{H}$ (that is, the graph with the same nodes as H but which contains an edge between nodes a and b iff H does not contain an edge between a and b). Let n' denote the number of nodes in H' and note that $n/2 < n' \leq n$ (for all sufficiently large n). The graph H' has n' nodes and at least

$$\frac{n'^2 - n'}{2} - \frac{n^2}{64y^2} \geq \frac{n'^2 - n'}{2} - \frac{n'^2}{16y^2}$$

edges. Therefore, it follows from Theorem 4.1 that H' contains a clique of $y^2 + 1 > x$ nodes. Let S be any set of x nodes from such a clique. For each pair of nodes a and b in S, the edge (a, b) is present in H', so the edge (a, b) is not present in H, which implies that a and b have outer regions that are at least $x^2 = \omega(x)$ apart in the graph G. As a result, the sets S and C satisfy Properties 1 and 2. □

THEOREM 4.3. *There does not exist a protocol with constant storage per process for messages and message headers that performs a constant amount of communication per* SEND *or* NRECV *posted by the application.*

Proof. Assume for the sake of contradiction that such an efficient protocol exists. Consider any set of n processes, let x, S, and C be as defined in Lemma 4.2, and for each node a, let a's *outer region* consist of the nodes in a's region that are not in C. Note that as a consequence of Property 2 in Lemma 4.2, if node a is in the outer region of node b in S and if node a' is in the outer region of node b' in S, where $b \neq b'$, then $a \neq a'$ and G does not contain the directed edge (a, a').

Now consider the execution of this protocol in which SENDs to process 1 are posted at each of the processes in S, each process not in C that received a posted SEND or a message from another process has sent all the messages that it is required to send by the protocol, and each message that has been sent has also been delivered, but no process in C has yet sent any messages and no NRECVs have yet been posted at process 1. Denote this point in the execution of the protocol time T.

At time T, the $o(x)$ processes in C can store only $o(x)$ headers and $o(x)$ data messages. Let S' be the subset of S consisting of each process a in S for which at time T there exists a process in C that contains either a's data message or a dynamic header for a node in a's outer region. Let C' be the set of processes consisting of the processes in C plus each process that is in the outer region of a process in S'. Note that at time T, all of the dynamic headers in C' are either for processes in C' or for processes that do not contain any dynamic headers. Let C'' be the set of processes consisting of the processes in C' plus each process a for which at time T there exists a process in C' with a header for a. Note that C'' contains $o(x)$ processes, that it contains process 1, and that all of the dynamic headers in C'' are for processes in C''. Also note that all of the processes not in C'' (in fact, all of the processes not in C) will not send any additional messages after time T until they receive at least one message. But there must exist at least one process s (and in fact, at least $x - o(x)$ processes) that is in $S \setminus C''$. The only processes that at time T contain the data from the SEND that was posted at s are the processes in s's outer region. However, all of these processes will remain inactive until they receive an additional message. Such an additional message must be generated (directly or indirectly) by one of the processes in C'', but any chain of messages initiated by a message in C'' must consist of $\omega(x)$ messages (as a result of Properties 1 and 2 in Lemma 4.2 and the fact that C'' only contains dynamic headers for nodes in C''). Thus the protocol will generate $\omega(x)$ messages, which is a contradiction. □

5. Upper bound for nonselective receives.

In this section, we present a protocol for SEND and NRECV that requires only constant storage per process and that performs a logarithmic amount of communication per SEND or NRECV posted by the application. This protocol is based on a deadlock-free packet-routing algorithm for a constant-degree, point-to-point network. We will first define a class of deadlock-free packet-routing algorithms for point-to-point networks called *buffer-reservation algorithms*, and we will show how buffer-reservation algorithms can be simulated using the hardware model presented in §2.1. We will then show how buffer-reservation algorithms can be used to develop protocols for nonselective receives, and we will examine one such buffer-reservation algorithm and the protocol that it produces.

5.1. Buffer-reservation algorithms.

Given a point-to-point network M consisting of n nodes, each of which contains a set of buffers, a *buffer-reservation* packet-routing algorithm specifies to which buffers a packet may be moved based solely on the packet's source node, destination node, and current buffer. More formally, a buffer-reservation algorithm A is a function $A(s, d, b) \to w$, where s and d are the packet's source and destination nodes, b is the packet's current buffer, and the set w (called the packet's *waiting set*) is the set of buffers to which the packet may be moved. All of the buffers in a waiting set must either be in the node that currently holds the packet or in neighboring nodes (that is, nodes that are connected by an edge to the node currently holding the packet). Each node in the network contains one *injection buffer* and one *delivery buffer*. Injection buffers are never allowed to appear in waiting sets, and the waiting set of a packet that is in a delivery buffer must be empty.

A standard technique for proving that a buffer-reservation algorithm is deadlock-free is to provide a total ordering of the buffers and to show that every packet that is not in the delivery buffer of its destination node has a waiting set that contains at least one buffer with higher rank than the buffer currently holding the packet [6, 7, 17]. We will call a buffer-reservation algorithm for which such an ordering of the buffers exists an *ordered buffer-reservation algorithm.*

We will prove the following theorem.

THEOREM 5.1. *Let c be a constant, let $f(n)$ be a function, and let A be an ordered buffer reservation algorithm for an n-node point-to-point network M with the following properties:*

1. *The degree of M is at most c.*
2. *Each node in M contains at most c buffers.*
3. *Regardless of a packet's source and destination nodes, every path that the packet can take when using algorithm A has length at most $f(n)$ and ends at the delivery buffer in the packet's destination node.*

There exists a communication protocol A'' for SEND *and* NRECV *between n processes that requires only constant storage per process and that performs only $O(f(n))$ communication per* SEND *or* NRECV *posted by the application.*

Our approach will be to simultaneously simulate a constant number of separate buffer-reservation algorithms on the hardware defined in §2.1. Each such buffer-reservation algorithm will provide a virtual network with its own injection buffer and delivery buffer in each node. In each of these virtual networks, a packet can be routed from any node's injection buffer to any node's delivery buffer with $O(f(n))$ communication. The communication protocol A'' will use these virtual networks to implement the SEND and NRECV commands in a deadlock-free manner, while using only constant storage per process and performing only $O(f(n))$ communication per SEND or NRECV posted by the application.

We will begin by describing a protocol A' that uses the hardware defined in §2.1 to simulate a single ordered buffer-reservation algorithm A having the properties given in Theorem 5.1. The protocol uses three types of control messages, called REQ, GRANT, and DENY, and one type of data message, called DATA. The n processes of A' correspond to the n nodes of M, and each process has (at most c) buffers corresponding to the buffers in M. Associated with each buffer i are three flags, called Full(i), Granted(i), and SendBuf(i). Full(i) indicates the presence of a message in i, Granted(i) indicates that i has been granted to some data message, and SendBuf(i) indicates that the contents of i should be sent out when the process's outgoing FIFO queue has sufficient space. Also associated with each buffer i are three arrays of flags, called SendReq(i, j), SendDeny(i, j), and Requested(i, j). SendReq(i, j) indicates that a REQ message should be sent to buffer j from buffer i, SendDeny(i, j) indicates that a DENY message should be sent to buffer j from buffer i, and Requested(i, j) indicates that an unsatisfied REQ message has been sent from buffer j to buffer i.

Note that because M has degree at most c and each node in M has at most

c buffers, each buffer in M can receive messages from at most $c^2 + c$ buffers and can send messages to at most $c^2 + c$ buffers. Therefore, the arrays SendReq(i, j), SendDeny(i, j), and Requested(i, j) are of constant length. In addition to the buffers and their associated flags, each process has two flags, called Inject and Deliver, that indicate whether a new message is ready to be placed in the injection buffer and whether the message in the delivery buffer can be removed. Initially, all of the flags are cleared (i.e., given the value FALSE). Finally, each process also contains (a constant number of constant-size) counters so that it can satisfy requests for buffers and for the outgoing FIFO queue in a fair, round-robin manner.

Intuitively, whenever a DATA message is placed in some buffer i, the protocol sends REQ messages from i to each buffer j in the DATA message's waiting set. When an available buffer j receives a REQ message from buffer i, it allocates buffer j (by setting GRANTED(j)) and sends a GRANT message to i. If buffer i is still attempting to send the DATA message when the GRANT from j arrives, it sends the DATA message to j (see Figure 4). On the other hand, if i is not attempting to send a DATA message to j, i sends a DENY message to j so that j can be granted to some other DATA message (see Figure 5).

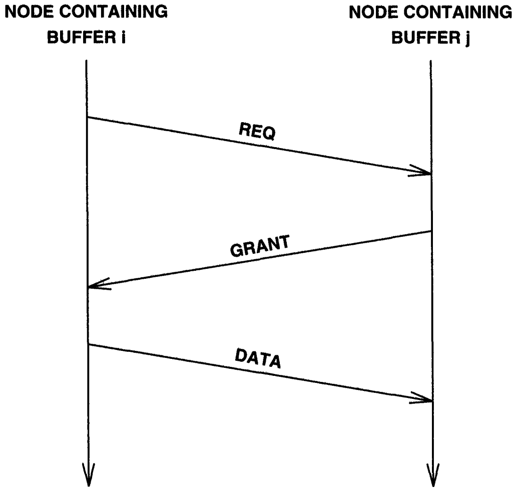


FIG. 4. *An execution of protocol A' in which a* DATA *message in buffer i is transferred to a buffer j in its waiting set.*

In the intuitive description above, there were cases in which the protocol was required to send a message in response to the receipt of a message. For example, the receipt of a DATA message destined for buffer i causes the protocoi to send REQ messages. However, in order to satisfy the continuous-consumption requirement, the protocol cannot delay the reception of the DATA message until the REQ messages can be sent. For this reason, the protocol uses the SendReq(i, j) flags to record its desire to send such REQ messages. When the the outgoing FIFO queue eventually becomes free, a SendReq(i, j) flag will be cleared and a REQ message from i to j will be placed in the outgoing FIFO queue. The SendDeny(i, j) flags play a similar role. In addition, flags are used to synchronize the placement of messages in injection buffers and the removal of messages from delivery buffers. When a new message is ready to be placed in an injection buffer, the corresponding Inject flag is set. Once the protocol has copied the new message into the injection buffer, the Inject flag is

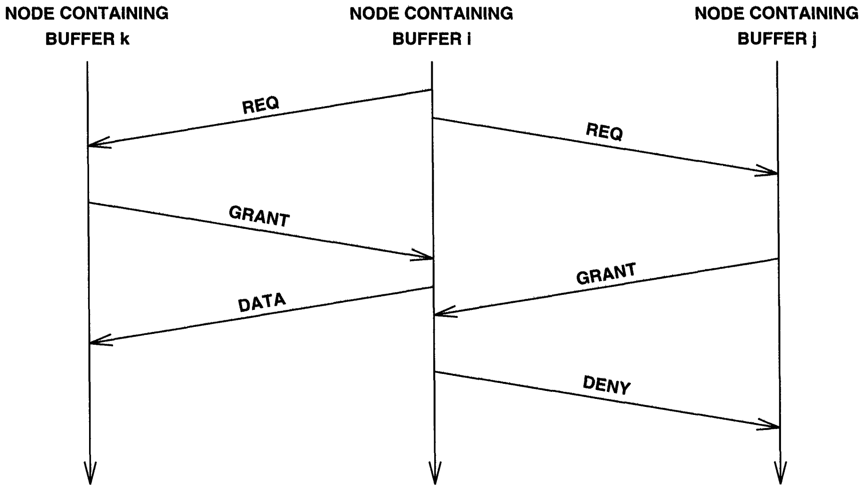


FIG. 5. *An execution of protocol A' in which a* DATA *message in buffer i is transferred to a buffer k in its waiting set. Buffer j is also in its waiting set.*

cleared. The Deliver flag plays a similar role with respect to the delivery buffer.

More formally, the protocol A' consists of three processes per node, called Inject/Deliver, ManageInFIFO, and ManageOutFIFO, each of which operates as a finite-state machine with a constant number of states. (However, note that it could also be viewed as the single process with a constant number of states obtained by taking the Cartesian product of these three finite-state machines.) Each process waits for certain events to occur, and when such an event occurs, it performs a fixed set of actions. The actions are performed atomically in the sense that all of the actions performed in response to one event are completed before the process performs any actions in response to a different event. The protocol operates as follows:

INJECT/DELIVER. This process responds to the following two events:

1. Either injection buffer i is empty and the Inject flag becomes TRUE or the Inject flag is TRUE and injection buffer i becomes empty. When this occurs, the new data message is copied to i, Full(i) is set, Inject and SendBuf(i) are cleared, and for each buffer j in the message's waiting set, SendReq(i, j) is set.

2. Either the delivery buffer i contains a data message and the Deliver flag becomes TRUE or the Deliver flag is TRUE and a data message is placed in delivery buffer i. When this occurs, the message is removed from the delivery buffer and Deliver and Full(i) are cleared. Then, if there exists a j such that Requested(i, j) is set, such a j is selected fairly, a GRANT message addressed to j is placed in i, Full(i), Granted(i), and SendBuf(i) are set, and Requested(i, j) is cleared.

MANAGEINFIFO. This process responds to the following four events:

1. A DATA message addressed to buffer i arrives in the incoming FIFO queue. When this occurs, the DATA message is moved from the incoming FIFO queue to i, Full(i) is set, Granted(i) and SendBuf(i) are cleared, and for each buffer j in the message's waiting set, SendReq(i, j) is set.

2. A REQ message from buffer j addressed to buffer i arrives in the incoming FIFO queue. When this occurs, the REQ message is removed from the incoming FIFO queue. If either Full(i) or Granted(i), Requested(i, j) is set. Otherwise, a GRANT message addressed to j is placed in i, Full(i), Granted(i), and SendBuf(i) are set, and

Requested(i, j) is cleared.

3. A GRANT message from buffer j addressed to buffer i arrives in the incoming FIFO queue. When this occurs, the GRANT message is removed from the incoming FIFO queue. If SendBuf(i) is FALSE and i contains a data message with j in its waiting set, the data message is addressed to j and SendBuf(i) is set. Otherwise, SendDeny(i, j) is set.

4. A DENY message addressed to buffer i arrives in the incoming FIFO queue. When this occurs, the DENY message is removed from the incoming FIFO queue. If there exists a j such that Requested(i, j) is set, such a j is selected fairly, a GRANT message addressed to j is placed in i, Full(i), Granted(i), and SendBuf(i) are set, and Requested(i, j) is cleared. Otherwise, Full(i) and Granted(i) are cleared.

ManageOutFIFO. This process responds either when the outgoing FIFO queue is empty and there is a SendReq, SendDeny, or SendBuf flag that becomes TRUE or when there is a SendReq, SendDeny, or SendBuf flag that is set and the outgoing FIFO queue becomes empty. When this occurs, it selects (fairly) a SendReq(i, j), SendDeny(i, j), or SendBuf(i) flag that is TRUE and does the following:

1. If SendReq(i, j) is selected, SendReq(i, j) is cleared and a REQ message from i to j is placed in the outgoing FIFO queue.

2. If SendDeny(i, j) is selected, SendDeny(i, j) is cleared and a DENY message from i to j is placed in the outgoing FIFO queue.

3. If SendBuf(i) is selected and i contains a GRANT message, SendBuf(i) and Full(i) are cleared and the GRANT message is placed in the outgoing FIFO queue.

4. If SendBuf(i) is selected and i does not contain a GRANT message, SendBuf(i) and Full(i) are cleared and the message in i is placed in the outgoing FIFO queue. Then, if there exists a j such that Requested(i, j) is set, such a j is selected fairly, a GRANT message addressed to j is placed in i, Full(i), Granted(i), and SendBuf(i) are set, and Requested(i, j) is cleared.

We will now prove that given any ordered buffer-reservation algorithm A having the properties given in Theorem 5.1, the protocol A' defined above performs deadlock-free communication from injection buffers to delivery buffers.

Lemma 5.2. *If X is a* DATA *or* GRANT *message stored in a buffer i, then X will not be overwritten by another message.*

Proof. First, note that if i is an injection buffer, X cannot be overwritten because the Inject/Deliver process only places DATA messages in i when i is empty, and i never has a GRANT message (because i is not in the waiting set of any message). Now consider the case where i is not an injection buffer. In this case, the Inject/Deliver process cannot overwrite X because it only places messages in the injection buffer. Also, the ManageInFIFO process cannot overwrite X because a DATA message addressed to i can only arrive after a GRANT message has been sent from i that indicates that i is empty, a GRANT message is only placed in i if Full(i) is FALSE, and REQ and DENY messages are never placed in buffers. Finally, the ManageOutFIFO process cannot overwrite X because it does not place messages in buffers. □

The following lemma proves that the protocol A' maintains the continuous consumption property.

Lemma 5.3. *If a control or data message X arrives in an incoming FIFO queue, it will be removed unconditionally from the incoming FIFO queue by the communication process within finite time.*

Proof. By the construction of A', the ManageInFIFO process removes message X from the incoming FIFO queue unconditionally and within finite time. □

LEMMA 5.4. *If a control or data message X that is addressed to buffer i is placed in an outgoing FIFO queue, it will arrive at the process containing buffer i within finite time.*

Proof. Recall that the network guarantees delivery, provided continuous consumption of messages by the communication processes. Because Lemma 5.3 guarantees continuous consumption of messages, any message placed in an outgoing FIFO queue will be delivered by the network within finite time. □

We will say that a message is *ready to be sent* if it is a DATA or GRANT message that is stored in some buffer i and SendBuf(i) is set or if it is a REQ or DENY message that is indicated by the fact that a SendReq or SendDeny flag is set.

LEMMA 5.5. *If a control or data message X is ready to be sent, it will be placed in the outgoing FIFO queue within finite time.*

Proof. It follows from Lemma 5.4 that any message currently in the outgoing FIFO queue will leave the outgoing FIFO queue in finite time. Because there are only c buffers, $c^3 + c^2$ SendReq flags, and $c^3 + c^2$ SendDeny flags competing for the same outgoing FIFO queue and these messages are serviced by ManageOutFIFO in a round-robin manner, at most $2c^3 + 2c^2 + c$ other messages can enter the outgoing FIFO queue before X enters it. From Lemma 5.4, each of these message will leave the outgoing FIFO queue in finite time. Furthermore, from Lemma 5.2, it follows that X will not have been overwritten. Therefore, the ManageOutFIFO process will place X in the outgoing FIFO queue. □

LEMMA 5.6. *If a control or data message X addressed to buffer i is ready to be sent, it will arrive at the process containing buffer i within finite time.*

Proof. This follows immediately from Lemma 5.5 and Lemma 5.4. □

LEMMA 5.7. *If a data message X is stored in a buffer i, then within finite time, either X will move to one of the buffers in its waiting set or every buffer j in X's waiting set will have its* Requested(j, i) *flag set.*

Proof. Consider any buffer j in the waiting set of X. When X was first placed in i, the process that placed X in i (either Inject/Deliver or ManageInFIFO) set the flag SendReq(i, j) and cleared the flag SendBuf(i). From Lemma 5.6, this causes a REQ message to arrive at the process containing buffer j within finite time. The ManageInFIFO process that receives this REQ message will either set the Requested(j, i) flag or place a GRANT message in j addressed to i. If it creates such a GRANT message, then from Lemma 5.6, the GRANT message will arrive at the process containing buffer i within finite time. Thus either Requested(j, i) is set or a GRANT from j arrives at i. If for every buffer j in X's waiting set, Requested(j, i) is set, the lemma holds. On the other hand, if there exists a buffer j in X's waiting set that causes a GRANT from j to arrive at i, let j' be the one that causes the first such GRANT message to arrive at i. When this GRANT message arrives, X will not have been overwritten (from Lemma 5.2) and SendBuf(i) will be set, so it follows from Lemma 5.6 that X will arrive at the process containing j' within finite time, at which point it will be placed in j'. □

LEMMA 5.8. *If a* GRANT *message in a buffer i is ready to be sent to a buffer j, then within finite time either a* DATA *message or a* DENY *message will arrive at i from j.*

Proof. From Lemma 5.6, the GRANT message will arrive at the process containing j within finite time. When this occurs, the ManageInFIFO process will either address a DATA message in j to i and set SendBuf(j) or it will set SendDeny(j, i). It follows from Lemma 5.6 that either a DATA message or a DENY message addressed to i

from j will arrive at the process containing i within finite time. □

At this point, we have shown that DATA messages will either advance within finite time or they will set all of the Requested(j, i) flags in the message's waiting set. Our next step will be to show that DATA messages will in fact always advance within finite time. However, in order to show this, we will have to assume that messages which arrive in delivery buffers are eventually removed so that other messages can make progress. In particular, we will require that the following *delivery-buffer-emptying* requirement is met: whenever a DATA message is placed in a delivery buffer, the associated Deliver flag is set within finite time, regardless of whether or not the injection buffer in the same node ever becomes empty. As a result, the message in the delivery buffer can be removed and other DATA messages can make progress. Note that if we did not have this delivery-buffer-emptying requirement, DATA messages might never be able to leave the network, thus causing deadlock. Also, note that the delivery-buffer-emptying requirement is identical to the continuous-consumption property, except that it applies at the level of the virtual network that is being simulated rather than at the level of the physical network.

LEMMA 5.9. *Assume that the delivery-buffer-emptying requirement is met. If a data message X is stored in a buffer i, then either i is a delivery buffer, in which case X will be removed from i within finite time, or i is not a delivery buffer, in which case X will move to one of the buffers in its waiting set within finite time.*

Proof. Assume for the sake of contradiction that the lemma does not hold, and let i be the highest-ranked buffer (according to the ordering of the buffers in A required by the definition of an ordered buffer-reservation algorithm) for which the lemma does not hold. If i is a delivery buffer, then it follows from the delivery-buffer-emptying requirement that the associated Deliver flag will be set within finite time. Furthermore, it follows from the construction of the Inject/Deliver process that X will be removed from i within finite time. However, this implies that the lemma holds for buffer i, which is a contradiction.

On the other hand, if i is not a delivery buffer, then it follows that there must exist a buffer j in X's waiting set with rank greater than i's rank. From Lemma 5.7, within finite time either X will move to a buffer in its waiting set (in which case the lemma holds) or the Requested(j, i) flag will be set. From the definition of i and Lemma 5.6, whenever j contains a data or control message, the message is moved out of j within finite time. Because the Requested(j, i) flag is set, a GRANT message will be placed in j and sent to some buffer k for which Requested(j, k) is set. From Lemma 5.6, this GRANT message will be delivered to k in finite time, and from Lemma 5.8, k will return a DATA or DENY message to j within finite time. Because there are a finite number of buffers which can request j and because these requests are satisfied in a round-robin manner, it follows that within finite time i will receive a GRANT from j. At this point, either X has already been addressed to another buffer in its waiting set and SendBuf(i) has been set or X will be addressed to j and SendBuf(i) will be set. In either case, from Lemma 5.6, X will be moved to a buffer in its waiting set within finite time. Therefore, the lemma holds for buffer i, which is a contradiction. □

5.2. Protocol for nonselective receives. Our protocol for SEND and NRECV is based on the *linked-list* protocol created by Burkowski, Cormack, and Dueck [3], which was discussed in §1. Recall that the linked-list protocol maintains a distributed linked list of the processes attempting to SEND to each process. This protocol requires only a constant amount of storage per process and it performs only a constant amount of communication per SEND or NRECV. Also, recall that the linked-list protocol does not satisfy the continuous-consumption requirement, so it cannot be implemented directly on the physical network. However, we will show that it can be implemented using just a contant number of virtual networks.

We will have to make two minor changes to the linked-list protocol. First, the original linked-list protocol [3] implemented a Reply primitive in addition to the SEND and NRECV primitives. Because we are only implementing SEND and NRECV primitives, we will only utilize the parts of the linked-list protocol that correspond to the SEND and NRECV primitives.[2] Second, in the original linked-list protocol [3], when a process is waiting in a linked-list of senders and it receives a request to send its data to the receiver, it must immediately send a message containing both its data and a pointer to the next process in the linked list.[3] Because in our implementation the pointer value may not be ready when the request to send the data arrives, we will use an extra flag in each process to record whether a request to send data to the receiver has arrived. We will call the linked-list protocol with these two minor changes the *modified linked-list protocol.*

The modified linked-list protocol uses five types of messages, called SEND_REQUEST, SEND_AGAIN, SEND_QUEUE, PROCEED, and REPLY_BLOCK. An example of an execution in which a process S_1 posts a SEND to a process R which has already posted a matching NRECV is shown in Figure 6. A SEND_REQUEST message containing the application message is sent from S to R. Process R responds by sending a REPLY_BLOCK message to S and unblocking its associated application process. When the REPLY_BLOCK message arrives at S, S unblocks its associated application process.

Another example of an execution of the modified linked-list protocol is shown in Figure 7. In that figure, S_1 and S_2 send SEND_REQUEST messages to R containing application messages. Because the matching NRECVs have not been posted when the SEND_REQUEST messages arrive, R discards the application messages and sends a SEND_QUEUE message to S_1 containing the address of S_2 (thus creating a linked list of the processes S_1 and S_2 waiting for R). When the first NRECV is posted at R, R sends a PROCEED message to S_1, which responds with a SEND_AGAIN message containing the application message and the address of S_2. Both S_1 and R know that the first message has been communicated successfully, so they unblock their associated application processes. When the next NRECV is posted at R, R sends a PROCEED message to S_2 (recall that R obtained the address of S_2 in the SEND_AGAIN message from S_1). S_2 then responds with a SEND_AGAIN message containing the application message and a null pointer (because S_2 is the last process in the linked list of waiting processes).

We will begin by assuming that each communication process has a single incoming FIFO queue and a single outgoing FIFO queue, each of which is of constant size, and that no outgoing FIFO queue remains full forever (regardless of the fact that the modified linked-list protocol does not satisfy the continuous-consumption requirement). Later, we will show how virtual networks can be used to guarantee that no outgoing FIFO queue remains full forever. The following properties of the modified linked-list protocol are immediate from the complete description of the linked-list protocol given

[2] Specifically, the Reply_Request and Error packets will not be used and the Reply_Blocked state will not exist.

[3] Specifically, when a Proceed packet arrives, the pointer value from the most recent Send_Queue packet must be sent along with the data.

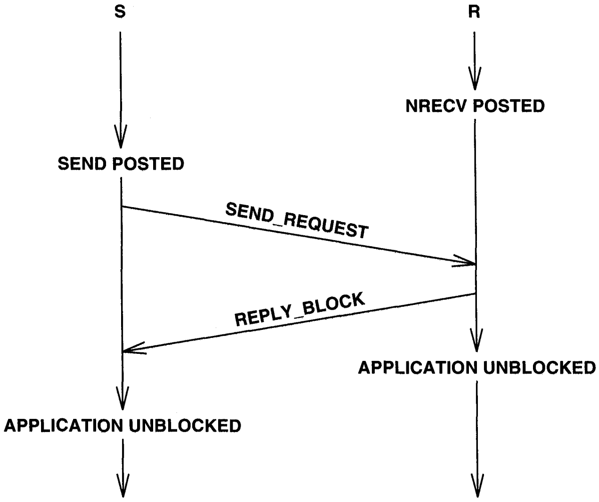


FIG. 6. *An execution of the modified linked-list protocol in which the* NRECV *is posted by* R *before the corresponding* SEND *is posted by* S.

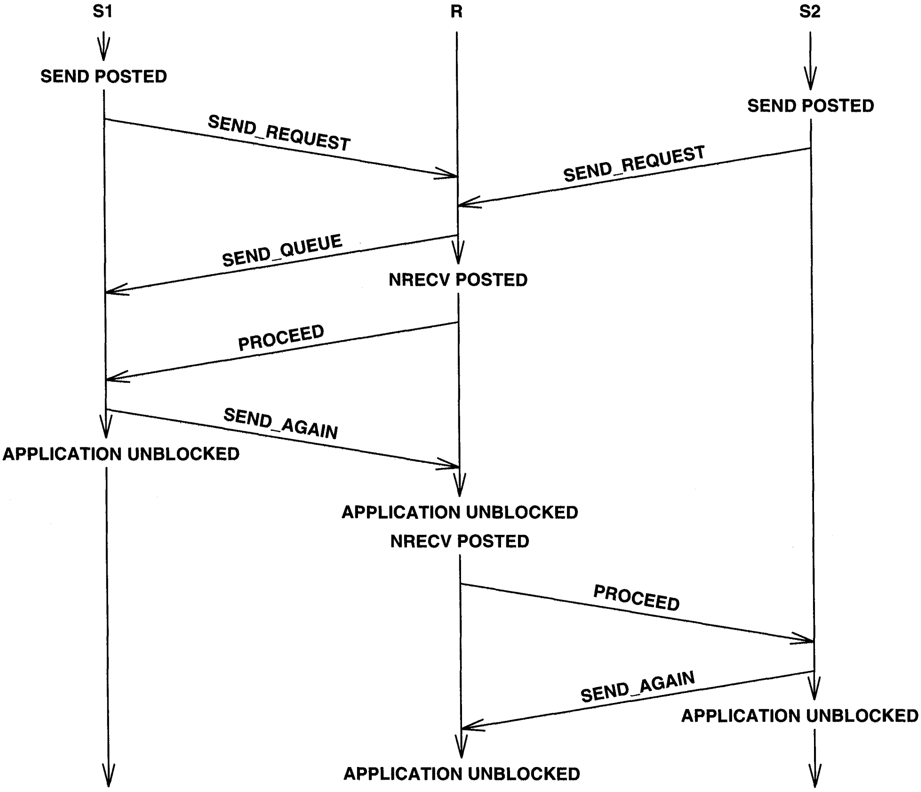


FIG. 7. *An execution of the modified linked-list protocol in which* SEND_REQUESTs *from* S_1 *and* S_2 *arrive at* R *before* R *posts the matching* NRECVs.

by Burkowski et al. [3].

THEOREM 5.10. *If each communication process has one incoming FIFO queue and one outgoing FIFO queue, each of constant size, the messages sent by the modified linked-list protocol can be divided into three classes*[4] *such that the following hold:*

1. *Whenever a process places a Class-*1 *message in its outgoing FIFO queue, no other Class-*1 *message is in the process's outgoing FIFO queue.*

2. *A Class-*2 *message is placed in an outgoing FIFO queue only in response to the removal of a Class-*1 *message from the incoming FIFO queue. Furthermore, no other messages are removed from the incoming FIFO queue or placed in the outgoing FIFO queue between the time the Class-*1 *message is removed from the incoming FIFO queue and the time the Class-*2 *message is placed in the outgoing FIFO queue.*

3. *Whenever a process places a Class-*3 *message in its outgoing FIFO queue, no other Class-*3 *message is in the process's outgoing FIFO queue.*

4. *Whenever a Class-*1 *message arrives in the incoming FIFO queue in process i, it is either removed unconditionally and within finite time or it is removed within finite time once the outgoing FIFO queue in process i has space for one additional Class-*2 *message.*

5. *Whenever a Class-*2 *or Class-*3 *message arrives in an incoming FIFO queue, it is removed unconditionally and within finite time.*

Furthermore, if no outgoing FIFO queue remains full forever, the modified linked-list protocol correctly implements SEND *and* NRECV *between n processes with only constant storage per process and a constant amount of communication per* SEND *or* NRECV *posted by the application.*

We are now prepared to prove Theorem 5.1.

Proof of Theorem 5.1. Given the buffer-reservation algorithm A, we utilize the simulation of A presented in §5.1 three times to create three separate virtual networks, each with its own injection and delivery buffers and its own Inject and Deliver flags. Then we implement the modified linked-list protocol by using the ith virtual network, $1 \leq i \leq 3$, to route Class-i messages. In particular, Class-i messages are sent by setting the ith Inject flag and waiting for them to be placed in the ith injection buffer and Class-i messages are received by copying them from the ith delivery buffer and setting the ith Deliver flag.

Lemma 5.9 shows that each virtual network is free of deadlock, provided that the delivery-buffer-emptying requirement is satisfied for that network. It follows from property 5 above that whenever a Class-2 or Class-3 message arrives in a delivery buffer, the associated Deliver flag will be set within finite time. As a result, the delivery-buffer-emptying requirement is satisfied for virtual networks 2 and 3, which are therefore free of deadlock. As a result, it follows that every message placed in a Class-2 injection buffer is removed within finite time.

Now let us consider virtual network 1. Property 4 above states that whenever a message is placed in a Class-1 delivery buffer, either the associated Deliver flag is set unconditionally and within finite time or it is set within finite time once the Class-2 injection buffer has space for one additional Class-2 message. However, we just showed that every message placed in a Class-2 injection buffer is removed within finite time. Therefore, even if the Class-2 injection buffer is full when a message is placed in a Class-1 delivery buffer, the Class-2 injection buffer will have room for an additional message within finite time, so the first Deliver flag will be set within finite time. As a

[4] Class 1 consists of the Send_Request messages, Class 2 consists of the Send_Queue and Reply_Block messages, and Class 3 consists of the Proceed and Send_Again messages.

result, virtual network 1 also satisfies the delivery-buffer-emptying requirement, and it follows from Lemma 5.9 that virtual network 1 is also free from deadlock.

Also, it follows from property 2 above that only Class-1 and Class-3 messages must be injected in response to an application SEND or NRECV. Therefore, it follows from properties 1 and 3 above that whenever a message must be injected in response to an application SEND or NRECV, the appropriate injection buffer will be free. As a result, the modified linked-list protocol will not deadlock, and the protocol is a correct implementation of SEND and NRECV.

Furthermore, it follows from the construction of the virtual networks that only constant storage is required per node. Finally, note that whenever a DATA message is placed in a buffer in a virtual network, it causes at most a constant number of Requested flags to be set, each of which causes at most a constant number of control and data messages to be sent. As a result, A'' performs only $O(f(n))$ communication per packet sent by the modified linked-list protocol. Because the modified linked-list protocol sends only a constant number of packets per SEND or NRECV posted by the application, it follows that A'' performs only $O(f(n))$ communication per SEND or NRECV posted by the application. □

The following theorem follows from a deadlock-free routing algorithm for the shuffle-exchange that was presented by Pifarré et al. [12].

THEOREM 5.11. *There exists an ordered buffer-reservation algorithm A for an n-node point-to-point network M such that the following hold:*

1. *M has degree at most* 3.
2. *Each node in M has at most four buffers.*
3. *Regardless of a packet's source and destination nodes, every path that the packet can take when using algorithm A has length at most* $3\log n$ *and ends at the delivery buffer in the packet's destination node.*

Combining Theorems 5.1 and 5.11 yields the following theorem.

THEOREM 5.12. *There exists a protocol for implementing* SEND *and* NRECV *among* n *processes that requires only constant storage per process and performs* $O(\log n)$ *communication per* SEND *or* NRECV *posted by the application.*

It should be noted that Theorem 5.1 gives a technique for generating a protocol A'' from an *adaptive* routing algorithm A. A simpler proof of Theorem 5.12 could be obtained by generating a protocol A'' from an *oblivious* routing algorithm A that requires constant storage per node and uses routes of length $O(\log n)$, such as the routing algorithm for a binary-tree network in which each node has two buffers, one for moving up the tree and one for moving down the tree [9]. However, this approach has the disadvantage of creating a bottleneck near the root of the tree.

REFERENCES

[1] Y. AFEK AND E. GAFNI, *End-to-end communication in unreliable networks*, in Proc. 7th Annual ACM Symposium on Principles of Distributed Computing, Association for Computing Machinery, New York, 1988, pp. 131–148.

[2] G. ANDREWS AND F. SCHNEIDER, *Concepts and notations for concurrent programming*, ACM Comput. Surveys, 15 (1983), pp. 3–43.

[3] F. BURKOWSKI, G. CORMACK, AND G. DUECK, *Architectural support for synchronous task communication*, in Proc. 3rd International Conference on Architectural Support for Programming Languages and Operating Systems, Association for Computing Machinery, New York, 1989, pp. 40–53.

[4] G. CHARTRAND AND L. LESNIAK, *Graphs and Digraphs*, 2nd ed., Wadsworth and Brooks/Cole Advanced Books and Software, Monterey, CA, 1986.

[5] I. Cidon and I. S. Gopal, *Control mechanisms for high speed networks*, in Proc. IEEE International Conference on Communications, IEEE Press, Piscataway, NJ, 1990, pp. 259–263.

[6] R. Cypher and L. Gravano, *Requirements for deadlock-free, adaptive packet routing*, SIAM J. Comput., 23 (1994), pp. 1266–1274.

[7] K. D. Günther, *Prevention of deadlocks in packet-switched data transport systems*, IEEE Trans. Comm., 29 (1981), pp. 512–524.

[8] C. Leiserson, Z. Abuhamdeh, D. Douglas, C. Feynmann, M. Ganmukhi, J. Hill, W. Hillis, B. Kuszmaul, M. S. Pierre, D. Wells, M. Wong, S.-W. Yang, and R. Zak, *The network architecture of the connection machine CM*-5, in Proc. 4th Annual ACM Symposium on Parallel Algorithms and Architectures, Association for Computing Machinery, New York, 1992, pp. 272–285.

[9] P. M. Merlin and P. J. Schweitzer, *Deadlock avoidance in store-and-forward networks* I: *Store-and-forward deadlock*, IEEE Trans. Comm., 28 (1980), pp. 345–354.

[10] ———, *Deadlock avoidance in store-and-forward networks* II: *Other deadlock types*, IEEE Trans. Comm., 28 (1980), pp. 355–360.

[11] Y. Ofek and M. Young, *Principles for high speed network control*, in Proc. 9th Annual ACM Symposium on Principles of Distributed Computing, Association for Computing Machinery, New York, 1990, pp. 161–175.

[12] G. Pifarré, L. Gravano, S. Felperin, and J. Sanz, *Fully-adaptive minimal deadlock-free packet routing in hypercubes, meshes, and other networks*, in Proc. 3th Annual ACM Symposium on Parallel Algorithms and Architectures, Association for Computing Machinery, New York, 1991, pp. 278–290.

[13] L. Pouzin, *Methods, tools, and observations on flow control in packet-switched data networks*, IEEE Trans. Comm., 29 (1981), pp. 273–286.

[14] G. Regnier, *Delta message passing protocol*, in Proc. 1st Intel Delta Applications Workshop, 1992, pp. 173–178.

[15] A. Sistla, *Distributed algorithms for ensuring fair interprocess communication*, in Proc. 3rd Annual ACM Symposium on Principles of Distributed Computing, Association for Computing Machinery, New York, 1984, pp. 266–277.

[16] A. S. Tanenbaum, *Network protocols*, ACM Comput. Surveys, 13 (1981), pp. 175–211.

[17] S. Toueg and K. Steiglitz, *Some complexity results in the design of deadlock-free packet switching networks*, SIAM J. Comput., 10 (1981), pp. 702–712.

[18] R. Watson, *The Delta-t transport protocol: Features and expansions*, in Proc. 14th Conference on Local Computer Networks, 1989, pp. 399–407.

[19] C. Williamson and D. Cheriton, *An overview of the VMTP transport protocol*, in Proc. 14th Conference on Local Computer Networks, 1989, pp. 415–420.

SIAM J. COMPUT.
Vol. 25, No. 5, pp. 1105–1121, October 1996

ON-LINE SCHEDULING OF IMPRECISE COMPUTATIONS TO MINIMIZE ERROR*

WEI-KUAN SHIH† AND JANE W. S. LIU‡

Abstract. This paper describes three algorithms for scheduling preemptive, imprecise tasks on a processor to minimize the total error. Each imprecise task consists of a mandatory task followed by an optional task. Some of the tasks are on-line; they arrive after the processor begins execution. The algorithms assume that when each new on-line task arrives, its mandatory task and the portions of all the mandatory tasks yet to be completed at the time can be feasibly scheduled to complete by their deadlines. The algorithms produce for such tasks feasible schedules whose total errors are as small as possible. The three algorithms are designed for three types of task systems: (1) when every task is on-line and is ready upon its arrival, (2) when every on-line task is ready upon arrival but there are also off-line tasks with arbitrary ready times, and (3) when on-line tasks have arbitrary ready times. Their running times are $O(n \log n)$, $O(n \log n)$, and $O(n \log^2 n)$, respectively.

Key words. real-time systems, scheduling to meet deadlines, deterministic scheduling, on-line scheduling

AMS subject classification. 68Q25

1. Introduction. The imprecise computation technique [1–10] has been proposed as a way to provide scheduling flexibility. In the imprecise computation model, each task consists of a mandatory portion followed by an optional portion. The mandatory portion must be executed to completion before the deadline of the task. The optional portion can be terminated before it is completed, if necessary, in order for the task and other tasks to meet their deadlines. The result produced by a prematurely terminated task contains an error that is a nondecreasing function of the processing time of the unexecuted portion. The scheduler ensures that the mandatory portions of all tasks are completed by the deadlines of the tasks while trying to keep the errors in their results small. Several efficient algorithms have been proposed to solve different scheduling problems, including the problem to minimize the total or average error [3–6] and the problem to minimize the maximal flow time [10].

In an earlier paper [5], we proposed an algorithm (called Algorithm **F**) that can find optimal schedules of imprecise, preemptable tasks with arbitrary ready times and deadlines on a uniprocessor system. Algorithm **F** is optimal in the sense that it always finds a feasible schedule, with the minimum average error, as long as feasible schedules of the tasks to be scheduled exist. Its time complexity is $O(n \log n)$. Here, by a *feasible schedule*, we mean one in which the mandatory portion of every task completes by its deadline. Algorithm **F** is also optimal in the aspect of complexity because the lower bound of finding an optimal schedule for this problem is $\Omega(n \log n)$. A drawback of Algorithm **F** is that it is an off-line algorithm; it cannot handle on-line scheduling. We say that an algorithm is *off-line* if the parameters of all the tasks to be scheduled are known before the algorithm is used to schedule the tasks and the execution of any task begins. Otherwise, if there are newly arriving tasks to be scheduled after the executions of some tasks begin, we say that scheduling is

* Received by the editors July 7, 1992; accepted for publication (in revised form) January 31, 1995. This research was partially supported by ONR contracts N00014-89-J-1181 and N00014-92-J-1146.

† Department of Computer Science, National Tsing Hua University, Hsinchu, Taiwan, Republic of China (wshih@cs.nthu.edu.tw).

‡ Department of Computer Science, University of Illinois at Urbana–Champaign, 1304 West Springfield Avenue, Urbana, IL 61801 (janeliu@cs.uiuc.edu).

on-line.

In this paper, we concentrate on on-line scheduling of imprecise tasks to minimize the total error of all tasks. We classify tasks as on-line and off-line. Tasks that arrive before the processor starts executing any task are *off-line tasks*, and tasks that arrive after the processor starts executing some tasks are *on-line tasks.* The parameters of an on-line task become known when it arrives. On-line scheduling is important for a real-time system. One of the most important functions for a real-time system is to handle external events. External events are usually unpredictable. An event can arrive at any time, and the parameters of the task handling the event can have arbitrary values. This unpredictable nature of the external events makes optimal on-line scheduling impossible [12]. In particular, it is known that there can be no on-line algorithm that is optimal in the following sense: it always finds, for any system of on-line imprecise tasks, a feasible schedule with the minimum total error whenever the system has feasible schedules. This paper describes three on-line algorithms for scheduling imprecise tasks preemptively on a processor to minimize the total error of all tasks. They are greedy in that they try to schedule as much of the optional potion of each task as possible after setting aside sufficient amounts of processor time for the mandatory portions of all the arrived tasks so that these tasks can complete on time. These algorithms are optimal, in the sense that they produce feasible schedules with the minimum total error, when the given task system satisfies the *feasible mandatory constraint.* We say that a task system satisfies this constraint if at the time of arrival of every new on-line task, its mandatory portion, together with the yet-to-be-completed mandatory portions of already arrived tasks, can always be precisely scheduled to complete by their deadlines. This constraint is often met by task systems in which the processing time of the mandatory portion of every task is small compared to the processing times of the optional portions.

The remaining of this paper is organized as follows. Section 2 gives a precise definition of the on-line imprecise-task-scheduling problem in general as well as the special case of the problem solved here. Sections 3–5 present our algorithms for scheduling on-line tasks to minimize total error. Section 6 is a summary and discusses future works.

2. On-line-scheduling problem. We are given a system of preemptable, imprecise tasks $\mathbf{T} = \{T_1, T_2, \ldots, T_n\}$ in which each task T_i is characterized by the following parameters, which are rational numbers:

(1) ready time r_i at which T_i becomes ready for execution;

(2) deadline d_i by which T_i must be completed; and

(3) processing time τ_i, which is the time required to execute T_i to completion in the traditional sense.

Logically, each task T_i is decomposed into two subtasks: the *mandatory* subtask M_i and the *optional* subtask O_i. Hereafter, we refer to M_i and O_i simply as tasks. We use M_i and O_i to mean specifically the mandatory task and the optional task of T_i, respectively, and use T_i to mean the task as a whole. Let m_i and o_i be the processing times of M_i and O_i, respectively, m_i and o_i are rational numbers, and $m_i + o_i = \tau_i$. The ready times and deadlines of the tasks M_i and O_i are the same as that of T_i, and O_i depends on M_i and therefore cannot begin execution until M_i is completed. In any valid schedule of $\mathbf{T}$, every task T_i is assigned at least m_i units of time; we say that M_i is *precisely scheduled.* If in a schedule, the task O_i is assigned σ_i units of processor time, the error ε_i of T_i is $o_i - \sigma_i$. The value of σ_i is in the range $[0, o_i]$; ε_i is zero if O_i is precisely scheduled, that is, $\sigma_i = o_i$. By the total error of a schedule,

	r_i	d_i	m_i	o_i	arrival time
T_1	0	10	0	6	0
T_2	0	14	4	0	0
T_3	2	10	4	0	2

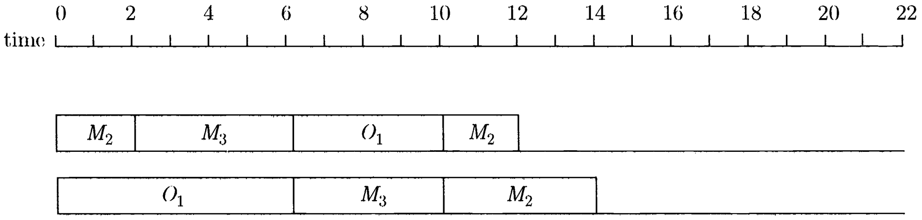


FIG. 1. *An example illustrating the difficulty in on-line scheduling.*

we mean the total error $\sum_i \varepsilon_i$ of all tasks when they are executed according to the schedule.

Earlier results on on-line scheduling (e.g., [12]) allow us to conclude that there is no optimal on-line algorithm that always finds a feasible schedule of an on-line task system whenever the given task system has feasible schedules. The example in Figure 1 illustrates that this fact remains to be true for imprecise task systems. In this example, we have three tasks. Tasks T_1 and T_2 are off-line tasks. Task T_3 is an on-line task and it arrives at time 2. Our goal is to keep the total error minimum, as well as to schedule all mandatory tasks so that they complete by their deadlines. A good strategy for achieving this goal is to arrange the executing order of mandatory and optional tasks so that the possibility of feasibly scheduling new on-line tasks is maximized. In other words, we execute mandatory tasks as soon as possible provided that we do not increase the total error by executing any mandatory task with a later deadline sooner than an optional task with an earlier deadline. According to this strategy, we choose to schedule T_2 at time zero. Although T_1 has earlier deadline than T_2, T_1 is not a mandatory task. We increase the possibility of feasibly scheduling M_2 and the mandatory tasks that might arrive in the future, but do not increase the total error, by executing T_2 in the interval $[0, 2]$. At time 2, T_3 arrives and is ready for execution. T_3 is a mandatory task with an earlier deadline than T_2. M_3 is scheduled at $t = 2$. The total error is 2. It is easy to see that we could have the feasible schedule with 0 total error shown in Figure 1 by scheduling T_1 in the interval $[0, 2]$. On the other hand, suppose that we schedule T_1 in the interval $[0, 2]$ but T_3 has a deadline at 14 and has processing time equal to 10. In this case, the task system cannot be feasibly scheduled, while it would be feasible if M_2 were scheduled in $[0, 2]$.

Because it is not possible to find optimal algorithms for scheduling general on-line imprecise-task systems, in the following three sections, we confine our attention to task systems that satisfy the feasible mandatory constraint. The algorithms described in these sections are optimal for such task systems. In §6, we will discuss their performance when used to schedule task systems that do not satisfy the feasible mandatory constraint.

We must consider four cases of the on-line scheduling problem. These cases are as follows.

Case 1. There is no off-line task, and every on-line task is ready for execution when it arrives.

Case 2. There is no off-line task, and on-line tasks have arbitrary ready times.

Case 3. There are off-line tasks, and every on-line task is ready for execution when it arrives.

Case 4. There are off-line tasks, and on-line tasks have arbitrary ready times.

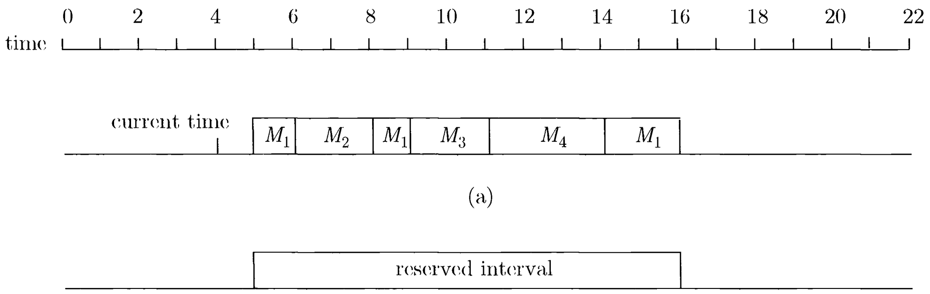
Cases 2 and 4 are the same in the following sense: in both cases, we face the problem of how to schedule the on-line tasks that are not ready for execution at the time instants of their arrivals. From the scheduling point of view, an on-line task that has arrived (and whose parameters have become known) but is not ready for execution can be viewed as an off-line task at the time when a new schedule is constructed. Therefore, we can eliminate Case 2 and consider the remaining three cases. The algorithms described in the next three sections are solution to Cases 1, 3, and 4.

3. On-line tasks ready upon arrival in the absence of off-line tasks. In this section, we present an algorithm for scheduling a system $\mathbf{T}$ of on-line tasks with arbitrary deadlines and execution times. In the case considered in this section, the ready time of each task is the instant of arrival of the task, and there is no off-line task. The algorithm, called the NORA (No Off-line tasks and on-line tasks Ready upon Arrival) algorithm, schedules all tasks so that their mandatory portions are completed by their deadlines and their optional portions are completed as much as possible. At any scheduling-decision time, tasks are scheduled on the earliest-deadline-first (EDF) basis. We will return to specify when scheduling decisions are made. The algorithm schedules every mandatory task M_i precisely by assigning to it m_i units of processor time. On the other hand, it may assign less than o_i units of time to the optional task O_i.

3.1. Reserving time for mandatory tasks. The NORA algorithm maintains a reservation list for all tasks that have arrived but are not yet completed and uses it as a guide in deciding where to schedule optional tasks and how much time to assign to them. A reservation list is derived from a feasible, precise schedule of the unfinished (that is, yet-to-be-completed) portions of all the mandatory tasks. We will describe in the next paragraph how this schedule is constructed. An interval that is assigned to a task in this schedule is a *reserved interval* in the reservation list; it is reserved for the execution of mandatory tasks. The reservation list tells us which intervals are reserved and which are not but does not distinguish the tasks to which the reserved intervals are assigned. The NORA algorithm never schedules any optional task in a reserved interval. In this way, the algorithm ensures that a sufficient amount of time is assigned to each mandatory task for it to complete by its deadline.

More precisely, a reservation list is defined by the time instants at which reserved intervals begin and end, as well as the amount of time reserved for each mandatory task. It is obtained from a precise schedule of all the unfinished portions of the mandatory tasks that have arrived, and this schedule is constructed using the reverse-scheduling approach. By *reverse scheduling*, we mean that all tasks are scheduled from the end of the reservation list, at the latest deadline, to the beginning, at the current time, in the following manner. Each task is scheduled at or before its deadline, and tasks are scheduled in the latest-ready-time-first order. Like the EDF algorithm, this strategy always finds a feasible schedule of all mandatory tasks whenever they have feasible schedules [11]. Figure 2 illustrates a feasible, precise schedule

	r_i	d_i	processing time of the unfinished portions
T_1	1	16	4
T_2	2	8	2
T_3	3	11	2
T_4	4	14	3



(a)

(the amounts reserved for $M_1 = 4$, $M_2 = 2$, $M_3 = 2$, and $M_4 = 3$)

(b)

FIG. 2. *An example of a reservation list.* (a) *The schedule produced by the reverse-scheduling algorithm.* (b) *The reservation list.*

constructed using the reverse-scheduling approach and the corresponding reservation list. In this example, we have four tasks. The table lists the parameters of the tasks, and the timing diagrams show the schedule and the reservation list produced by reverse scheduling of the tasks at time $t = 4$. We note that the roles of deadlines and ready times are reversed. We schedule these four tasks starting from time 16. The deadline of task M_1 is 16. From the reverse-scheduling point of view, it means that task M_1 is "ready" to be scheduled at 16. We schedule M_1 in the interval (14,16). At time 14, M_4 is "ready" to be scheduled, and its ready time is later. Therefore, M_4 preempts M_1 at time 14. We schedule M_4 in the interval (11,14). Because M_3's deadline is 11 and its ready time is later than the ready time of M_1, M_3 is scheduled in (9,11). We repeat the same process until all mandatory tasks are scheduled and produce the schedule in Figure 2(a). The corresponding reservation list maintained by the algorithm is shown in Figure 2(b).

The reservation list is updated each time a new task T_i arrives. The result is that some interval(s) before the deadline d_i of total length m_i is (are) reserved for the mandatory task M_i in order to ensure its completion before d_i. We will return later to describe the reservation step, which adds new reserved interval(s) into the reservation list upon arrivals of new tasks, as well as the times when reserved intervals are released, that is, made available for optional tasks.

3.2. Scheduling decisions. The NORA algorithm works as follows: the scheduler maintains a prioritized task queue in which tasks are ordered on the EDF basis. In the beginning, there is no task in the task queue, and the processor is idle. When

```
1. For as long as no event occurs, assign the task at the head of the task queue, the task with
   the earliest deadline, to processor for execution.
2. When an event occurs:
   2.1 Event_1: the current task is completed or terminated at its deadline:
       —remove the current task from the task queue;
       —cancel the reservation of the current task;
       —goto step 1.
   2.2 Event_2: the beginning of the first reserved interval is reached:
       —if there is time reserved for the current task,
            then cancel the reservation of the current task;
            else terminate the current task and remove it from the task queue;
       —goto step 1.
   2.3 Event_3: a new on-line task arrives:
       —update the reservation of the current task;
       —make reservation for this new task;
       —insert this task into the task queue;
       —goto step 1.
```

FIG. 3. *Pseudocode of the NORA algorithm.*

an on-line task arrives, the scheduler makes reservation for this new task, puts this task into the task queue, and schedules the first task in the task queue for execution. For every task, the processor always executes the mandatory portion first and then tries to execute the optional portion if there is enough time. Every task is terminated either when its optional portion completes or when its deadline is reached. When a task is terminated, it is removed from the task queue. A scheduling decision is made whenever any of the following possible events occurs. In the description of the events, we use t to denote the time at which an event occurs and T_i to denote the task executing at t.

Event_1. Task T_i (including M_i and O_i) completes or the deadline d_i of T_i is reached, that is, T_i terminates either normally or prematurely.

Event_2. The current time t reaches the beginning of a reserved interval in the reservation list.

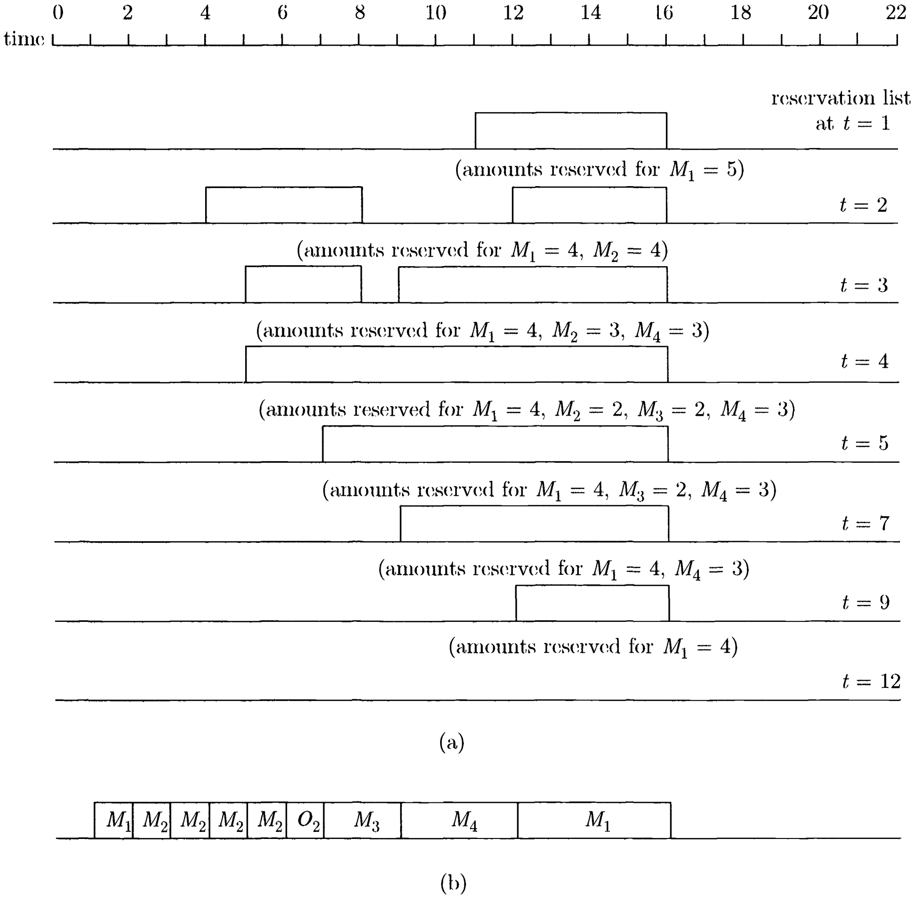
Event_3. A new on-line task arrives.

Figure 3 gives the pseudocode of the NORA algorithm. When an Event_1 occurs, any time x still reserved in the reservation list for the terminated task T_i is released. This is done by deleting the earliest interval(s) of length x from the reservation list. (We will return to show that this operation is correct.) The task at the head of the task queue with the earliest deadline among all the unfinished tasks is scheduled. The processor continues to execute the newly scheduled task until the next event occurs.

When an Event_2 occurs, the current time t is the beginning of a reserved interval. It is possible that some time remains to be reserved for M_i according to the reservation list. In this case, this reserved time (say x units) is released; the earliest interval(s) of length x is (are) deleted from the reservation list. We sometimes refer to this action as *canceling* the reservation of M_i. The processor continues to execute T_i. If, on the other hand, no time is reserved for T_i, T_i is terminated. The task with the earliest deadline among all the tasks that have reserved times according to the reservation list is scheduled and executed.

When an Event_3 occurs, the current executing task T_i is preempted, and the reservation list is updated. If the mandatory task M_i is completed at the time,

	r_i	d_i	t_i	m_i	o_i
T_1	1	16	8	5	3
T_2	2	8	6	4	2
T_3	4	11	3	2	1
T_4	3	14	5	3	2


0 2 4 6 8 10 12 14 16 18 20 22
time
reservation list at $t = 1$
(amounts reserved for $M_1 = 5$)
$t = 2$
(amounts reserved for $M_1 = 4$, $M_2 = 4$)
$t = 3$
(amounts reserved for $M_1 = 4$, $M_2 = 3$, $M_4 = 3$)
$t = 4$
(amounts reserved for $M_1 = 4$, $M_2 = 2$, $M_3 = 2$, $M_4 = 3$)
$t = 5$
(amounts reserved for $M_1 = 4$, $M_3 = 2$, $M_4 = 3$)
$t = 7$
(amounts reserved for $M_1 = 4$, $M_4 = 3$)
$t = 9$
(amounts reserved for $M_1 = 4$)
$t = 12$
M_1 M_2 M_2 M_2 M_2 O_2 M_3 M_4 M_1

(a)

(b)

FIG. 4. *An example illustrating the NORA algorithm.* (a) *The reservation list.* (b) *The final schedule.*

the time reserved for it is released. If M_i requires x units more of processor time to complete, we update its reservation by adding or deleting a reserved interval in the beginning of the reservation list so that the total reserved time for M_i is x. A reserved interval (or intervals) of length equal to the processing time of the newly arrived mandatory task is inserted into the reservation list. After the reservation list is updated, the task with the earliest deadline is scheduled and executed if the arrival time t is not in a reserved interval. Otherwise, the step taken when an Event_2 occurs is carried out.

The operations of the NORA algorithm are illustrated by the example in Figure 4. The table in Figure 4 gives the total execution times of all mandatory and optional tasks. No task arrives after $t = 4$. Figure 4(a) shows the reservation lists generated at different time instants at which the reservation list is updated. Figure 4(b) shows the schedule produced by the algorithm. At $t = 1$, T_1 arrives. The interval (11,16) is reserved for it, and it is scheduled for execution. At $t = 2$, T_2 arrives. The reservation list is updated. T_1 is preempted, and T_2 is scheduled because T_2 has an earlier deadline. At $t = 3$ and $t = 4$, T_4 and T_3 arrive. The reservation list is updated each time. T_2 continues to execute until $t = 5$ when the beginning of the reservation list is reached. Since there are still two units of time reserved for M_2, this time is released, and T_2 continues to execute. At $t = 7$, the beginning of the reservation list is reached again. T_2 is terminated at this time. T_3 is at the head of the task queue and has reserved time according to the reservation list; it is scheduled. Similarly, T_3 is terminated and T_4 and T_1 are scheduled at $t = 9$ and $t = 12$, respectively, to produce the schedule in Figure 4(b). The total error is 7.

3.3. Updating the reservation list. We must reserve m_j units of time for each newly arrived on-line task T_j. This is a critical part in the NORA algorithm. Clearly, we can construct a new reservation list from the beginning using the reverse scheduling algorithm each time a new task arrives. This approach will give us an algorithm with run time $O(n^2 \log n)$. Fortunately, we can update the reservation list quickly when making new reservations.

To see how, we note that it is not necessary to move the reserved interval(s) to different places during such an update. We can simply reserve for the newly arrived mandatory task M_j the interval(s) prior to the deadline of T_j that is (are) not reserved in the old list. This reservation can be done by reverse scheduling M_j alone in the gaps between the reserved intervals in the old reservation list. We call this action *inserting* M_j into the old list. For example, suppose that at $t = 2$, task T_3 in Figure 5 arrives. We insert M_3 into the old list. Because the deadline of T_3 is 9 and the interval [6,8] is already reserved, we reserve the intervals [8,9] and [5,6] for M_3. Similarly, at $t = 3$ when T_4 arrives, we simply insert M_4 into the old list, adding the reserved intervals [4,5] and [9,11].

Adding new reservations to the reservation list by simply inserting the new tasks into the old list is possible because the NORA algorithm only checks where reserved intervals begin. It does not keep track of which interval is reserved for what task. This means that we can construct the reservation list in any way as long as the beginnings of the reserved intervals in the resultant reservation list are the same as those in the reservation list constructed by reverse scheduling all the unfinished mandatory tasks together with the new mandatory task. The following lemma shows that we can indeed construct the new reservation list in the faster way described above.

LEMMA 3.1. *The new reservation list produced by simply inserting the newly arrived mandatory task M_j into the old list is the same as the reservation list derived from the schedule that is produced by using the reverse-scheduling algorithm on all unfinished mandatory tasks.*

Proof. To simplify our discussion, let $S_1(n)$ and $S_2(n)$ denote the reservation lists produced by simply inserting the nth new mandatory task into the old list $S_1(n-1)$ and the list produced by using the reverse scheduling algorithm on all unfinished mandatory tasks upon the arrival of the nth task, respectively. We want to prove this lemma by induction. Clearly, $S_1(1)$ and $S_2(1)$ are the same. Now suppose that $S_1(n-1)$ and $S_2(n-1)$ are the same but $S_1(n)$ and $S_2(n)$ are different. We compare the two lists $S_1(n)$ and $S_2(n)$. In the schedule produced by using the reverse-scheduling algorithm on all unfinished mandatory tasks and the new task, all the tasks

	r_i	d_i	m_i
T_1	1	16	4
T_2	2	8	2
T_3	3	9	2
T_4	4	11	3

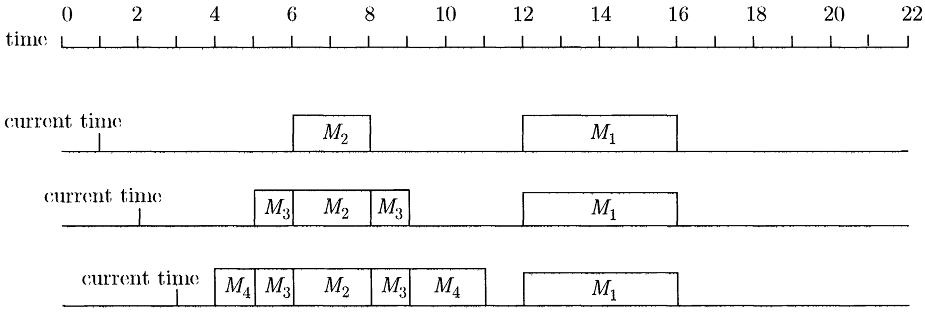


FIG. 5. *An illustrative example of updating the reservation list.*

are scheduled as late as possible. The gaps between reserved intervals in $S_2(n)$ cannot be moved earlier because the reserved intervals in $S_2(n)$ cannot be moved later. Since $S_1(n)$ and $S_2(n)$ are different, $S_1(n)$ contains at least one reserved interval that can be moved to a later time. However, $S_1(n)$ is constructed by inserting the nth task into the old reservation list $S_1(n-1)$ according to the reverse-scheduling algorithm. Since $S_1(n-1)$ is the same as $S_2(n-1)$, all reserved intervals in $S_1(n-1)$ cannot be moved to later times, and no gap in $S_1(n-1)$ can be moved earlier. Because the nth task is scheduled in the gaps in $S_1(n-1)$, no reserved interval in $S_1(n)$ can be moved later. This is a contradiction to the supposition that $S_1(n)$ and $S_2(n)$ are different. We can thus conclude that $S_1(n)$ and $S_2(n)$ are the same. □

Now that we know how to add the reservation of a new task into the reservation list, the remaining problem is how to cancel the reservation of a task and release the time reserved for the task. Again, the most straightforward way to do this is to apply the reverse-scheduling algorithm to construct a new reservation list each time. Again, we are fortunate and do not need to use this time-consuming method. According to the NORA algorithm, the reservation of a task is canceled only when it is the currently executing task. We cancel the reservation of the currently executing task by deleting the earliest reserved interval(s) in the reservation list. The following lemma shows that this simple method updates the reservation list correctly since the currently executing task is the one with the earliest deadline.

LEMMA 3.2. *If all unfinished tasks are ready, the reservation of the task with the earliest deadline for x units of time can be canceled by deleting the reserved interval(s) of length x from the beginning of the reservation list.*

Proof. The task with the earliest deadline is denoted by T_i. Given a schedule produced by reverse scheduling all the unfinished tasks, we can move the intervals assigned to T_i to the beginning of the schedule by swapping the time intervals assigned to T_i with other tasks that have later deadlines than it. Therefore, we can consider the reserved interval(s) in the beginning of the reservation list as being reserved for

T_i and delete them when we want to cancel the reservation for T_i. □

3.4. Optimality of the NORA algorithm. The optimality of the NORA algorithm requires that the following two conditions are satisfied:

1. All mandatory tasks are completed before their deadlines.
2. The total error is minimized.

The satisfaction of condition 1 for any task system that satisfies the feasible mandatory constraint discussed in §2 is obvious. The algorithm always reserves sufficient time for all the unfinished mandatory tasks before their deadlines. The reservation for a task is canceled only when we have completed the task. Theorem 1 shows that condition 2 is satisfied and the NORA algorithm is optimal whenever the task system satisfies the feasible mandatory constraint.

THEOREM 3.1. *The* NORA *algorithm can find a schedule with the minimum total error for a task system consisting solely of on-line tasks that are ready upon arrival.*

Proof. To show that any schedule found by the NORA algorithm has the minimum total error, we examine the sources of the errors. From the pseudocode of NORA in Figure 3, there are three kinds of events. When an Event_1 occurs, the current task is terminated because it is completed or its deadline is reached. The error in its result when it terminates is as small as possible. When an Event_3 occurs, no task is terminated. Therefore, no error is produced in step 2.3. An Event_2 occurs when we reached the beginning of the first reserved interval. In step 2.2, some error is produced when we terminate the current executing task and remove it from the task queue if it is not completed at the time. However, this optional task is not completed because there is no time to complete it. In other words, at least one of the unfinished mandatory tasks would miss its deadline if we continue to execute this optional task. Moreover, because no intentional idle time is inserted in the schedule, we cannot execute other tasks with earlier deadlines than the current task earlier to make room for this optional task without increasing their errors. Consequently, the error generated by step 2.2 in handling an Event_2 is minimal. □

4. On-line tasks with arbitrary ready times together with off-line tasks.

If we are satisfied with an $O(n^2 \log n)$ algorithm for scheduling on-line tasks that have arbitrary ready times in the presence of off-line tasks, we can easily find one by modifying the NORA algorithm in a straightforward manner. An example is the algorithm described by the pseudocode in Figure 6. Whenever the task system satisfies the feasible mandatory constraint, this algorithm is optimal in the sense that it guarantees all mandatory tasks are completed by their deadlines and the total error is minimized. The following theorem states this fact; its proof follows straightforwardly from the proof of Theorem 1.

THEOREM 4.1. *The algorithm in Figure* 6 *can find a schedule with minimum total error for an on-line task system in which on-line tasks have arbitrary ready times and there are off-line tasks.*

Because tasks have arbitrary ready times, we must keep track of where the reserved intervals of the individual tasks are. The reservation list is the schedule generated by reverse scheduling all the unfinished tasks. The complicated reverse-scheduling step must be repeated each time to insert a new task into the schedule or to delete a task from the schedule. In this section, we present techniques that lead to algorithms with run times $O(n^2)$ and $O(n \log^2 n)$.

4.1. Reservation array. To speed up the algorithm in Figure 6, we need a data structure other than the reservation list to tell us when mandatory tasks must

```
0. Generate the reservation list of all off-line tasks.
   Put all ready tasks in the task queue ordered on the EDF basis.
1. For as long as the task queue is not empty and no event occurs, assign the task at the head
   of the task queue, the task with the earliest deadline, to the processor for execution.
2. When an event occurs:
   2.1 Event_1: a task becomes ready:
       —update the reservation of the current task;
       —put the newly ready task in the task queue;
       —goto step 1.
   2.2 Event_2: the mandatory portion of the current task completes:
       —cancel the reservation of the current task from the reservation list;
       —goto step 1.
   2.3 Event_3: the optional portion of the current task completes or terminated at its
       deadline:
       —remove the task from the task queue;
       —goto step 1.
   2.4 Event_4: a new on-line task arrives:
       —if the current task is mandatory, update the reservation of the current task;
       —make reservation for this new task;
       —if this task is ready, insert it into the task queue;
       —goto step 1.
   2.5 Event_5: the beginning of a reserved interval is reached:
       —if the interval is reserved for the current task,
             then goto step 1;
             else update the reservation of the current task;
             —put the task assigned to the reserved interval at the head of the task queue;
       —goto step 1.
```

FIG. 6. *An algorithm for scheduling tasks with arbitrary ready times.*

be scheduled and which mandatory task to schedule so that all mandatory tasks are sure to complete by their deadlines. For this purpose, we use an array, called the *reservation array.* It is denoted by P. There is an element in this array for every task whose parameters are known. In particular, let σ_j be the processing time of the portion of the mandatory task M_j that is completed when we update the reservation array. (Later, we will discuss when this array is updated.) The value of the reservation array element $P(i)$ corresponding to the task T_i, which is either an off-line task or an arrived on-line task, is given by

$$P(i) = d_i - \sum_{d_j \le d_i} (m_j - \sigma_j). \tag{1}$$

It is the difference—the slack—between the deadline d_i and the total amount of processor time that must be assigned to the unfinished portions of all the mandatory tasks whose deadlines are equal to or less than d_i. Because the feasible mandatory constraint is satisfied, $P(i)$ is larger than or equal to zero for all i at all times.

At the beginning, there is no on-line task. We construct the reservation array P for all the off-line tasks. σ_j is zero for all j before the processor starts executing. The initial value of $P(i)$ is $d_i - \sum_{d_j \le d_i} m_j$ for all off-line tasks T_i. After the processor starts executing, we add offsets—that is, increments in processing times of the completed portions of mandatory tasks—to all elements of P to reflect the progress made toward the completion of the mandatory tasks. When a new on-line task T_l arrives, we add a new element $P(l)$, and its initial value is given by equation (1), where σ_j is the processor time of the completed portion of M_j at the time when T_l arrives.

To explain how the information provided by the reservation array can be used to determine when (and which) mandatory tasks must be scheduled, we again examine the algorithm in Figure 6. When the current time t reaches the beginning of a reserved interval in the reservation list, there exists a deadline, denoted by d_b, which is such that the entire interval between d_b and the current time is reserved. We call this deadline d_b the *bottleneck* of the task system at the current time. The processor can keep on executing optional tasks as long as there is no bottleneck. When a bottleneck exists, the processor must execute some unfinished mandatory tasks; otherwise, at least one mandatory task will miss its deadline. At any time t, we can determine whether a bottleneck exists and which deadline is the bottleneck from the value of $P(i)$. In particular, a deadline d_i becomes a bottleneck when it satisfies the inequality

$$d_i - \sum_{d_j \leq d_i} (m_j - \sigma_j) \leq t,$$

that is, when $P(i) \leq t$.

In the example shown in Figure 7, we have three off-line tasks. Initially, $P(1) = 5$, $P(2) = 4$, and $P(3) = 2$. At time 1, there is no bottleneck, as indicated by the values of $P(i)$ and illustrated by the timing diagrams in Figure 7(a). At time 2, we have completed one unit of M_1, $\sigma_1 = 1$, and the array P is updated. Both d_2 and d_3 are less than d_1. We only need to add one to $P(1)$. At this time, $P(3)$ is equal to t. We find a bottleneck d_3 at time 2. It means that no optional task can be scheduled in the interval from 2 to d_3. This fact is illustrated by the timing diagrams in Figure 7(b).

We now modify the algorithm in Figure 6, making use of the reservation array P instead of the reservation list in order to speed up the algorithm. This array is generated and initialized as described above. All the ready tasks are placed in the task queue and ordered on the EDF basis. At time zero, we schedule the task with the earliest deadline among all ready tasks. We keep on executing the current task until one of the following five events occurs.

(1) Event_1: *Some task becomes ready for execution.* When such an event occurs, we record the change in the amount of processor time required to complete the mandatory task of the current task in the reservation array P. We only need to modify the elements in P for the tasks with deadlines equal to or later than the deadline of the current task. Let the current task be T_j. The new offset is the processing time of the portion of M_j that is completed since the last update of the array P and is denoted by $\Delta\sigma_j$. The formula to modify elements of the array P is $P(k) = P(k) + \Delta\sigma_j$ if $d_k \geq d_j$; otherwise, $P(k)$ remains unchanged. After updating the array P, the newly ready task is put on the task queue, and the task with the earlier deadline is scheduled and executed.

(2) Event_2: *The mandatory task of the current task completes.* We modify the array P as we do in (1), when an Event_1 occurs, and choose the task at the head of the task queue for execution.

(3) Event_3: *The optional portion of the current task completes or is terminated at its deadline.* In this case, we remove the current task from the task queue and select the next task from the task queue for execution.

(4) Event_4: *A new on-line task arrives.* In this case, we need to modify the array P as follows. We first update the array P according to (1). An empty element

	r_i	d_i	m_i
M_1	1	18	4
M_2	2	8	4
M_3	4	11	5

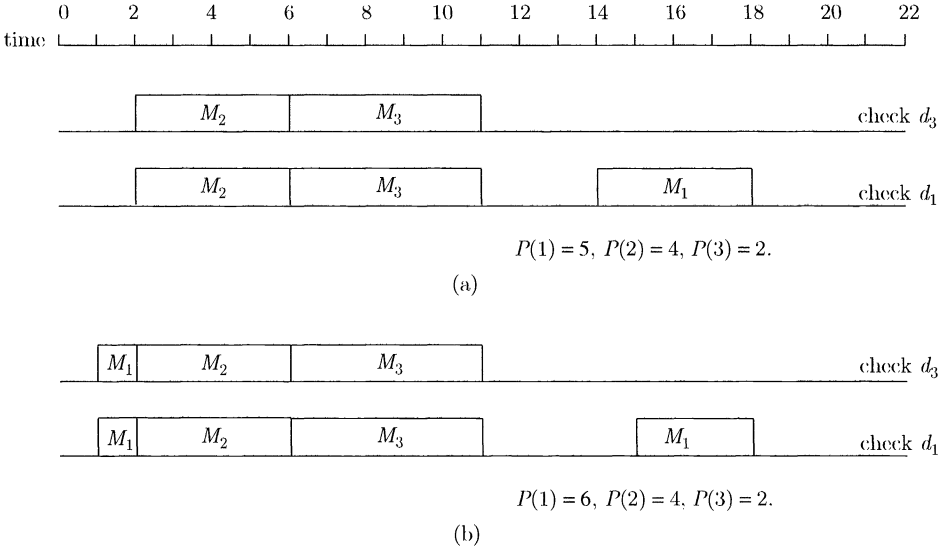


FIG. 7. *An example illustrating the usage of the array P.* (a) *When $t = 1$, there is no bottleneck.* (b) *When $t = 2$, d_3 becomes a bottleneck.*

is allocated to the new task and is used to store information about the task. Let the new task be T_l and the new element be $P(l)$. We find the largest deadline, denoted by $d_{l'}$, that is less than d_l and assign the content of $P(l')$ to $P(l)$. Then we add $d_l - d_{l'} - m_l$ to $P(l)$ and subtract m_l from all elements $P(k)$, where $d_k > d_l$. After the array P is updated, we add the task T_l into the task queue if it is ready and schedule the task at the head of the task queue.

(5) Event_5: *Some deadline becomes a bottleneck.* Let the deadline that is the bottleneck be d_j. We update the array P as in (1). If d_j is no longer a bottleneck after the update, we continue to execute the current task. If d_j remains a bottleneck, this means that all the time from the current time to d_j is reserved for mandatory tasks and the processor does not have time to execute any optional tasks in this interval. The ready mandatory tasks are scheduled on the EDF basis in the interval between the current time and d_j.

This algorithm, called the OAR (On-line tasks with Arbitrary Ready time) algorithm, is simple. It contains $O(n)$ steps. Here n is the number of tasks in the task system, including on-line and off-line tasks. The run time for each step is at most $O(n)$. Therefore, the algorithm has run time $O(n^2)$.

4.2. Speed-up method. The OAR algorithm described above is slow because we spend $O(n)$ to modify the consecutive elements of the reservation array each time we update the array. To speed up this step, we use a hierarchy of priority queues instead of an array to store the elements $P(i)$. We can modify any specific group of elements in the queue in $O(\log^2 n)$ time.

We begin by using a binary tree to construct the priority queue. The tree is constructed based on the deadlines of all off-line tasks plus a nonexisting deadline which is later than all deadlines of off-line and on-line tasks. The leaves of the binary tree represent the deadlines. Without loss of generality, we assume that the number u of off-line tasks plus one is a power of two. We construct a complete binary tree based on these $u + 1$ deadlines such that every internal node of the binary tree represents several consecutive deadlines. The leaf node $P(i)$ representing the deadline d_i has the value $d_i - \sum_{d_j \leq d_i} m_j$, the initial value of $P(i)$. The value of each internal node of the binary tree at any time is the total offset that applies to all leaves of the subtree rooted at this internal node. Because this is a priority queue, each internal node also has a pointer showing which leaf in the subtree rooted at it has the smallest value after all offsets are added to them. Therefore, the task system will have a bottleneck when the leaf pointed by the root of the binary tree has a value less than or equal to t. The following operations update the values of the nodes in the binary tree.

(1) To add an offset to all leaves, we add the offset to the root of the whole tree.
(2) To add an offset to all leaves that have deadlines larger than some specified deadline d_j, we find the path from the root to the leaf node that corresponds to the deadline d_j. Then we examine all nodes on this path. If an examined node is the left child of a node on this path, we find its right sibling and add the offset to this right sibling.

The run time of these two operations is at most $O(\log u)$. The time to find the smallest value from this priority queue is at most $O(\log u)$ also. Therefore, if there are no on-line tasks, the run-time of the OAR algorithm is $O(u \log u)$.

When on-line tasks arrive, the binary tree of the priority queue must be expanded to accommodate the elements of these on-line tasks. To avoid reconstructing the whole binary tree, we simply insert the node representing the new deadline into the tree upon the arrival of each new task. This operation will destroy the completeness of the binary tree; the longest path of the binary tree will no longer be $O(\log n)$, where n is the total number of tasks in the task system. To shorten the length of the longest path, we construct a hierarchy of priority queues and use these queues to store the reservation array elements of all on-line tasks. We assign a hierarchy of priority queues to each leaf of the main priority queue, the queue constructed based on the deadlines of the off-line tasks. In each hierarchy, there are at most $O(\log n)$ layers. In each layer, there is at most one priority queue. Therefore, the total number of priority queues in each hierarchy is at most $O(\log n)$. The number of nodes in the ith layer is equal to 2^{i-1}.

When an on-line task arrives, we find in the main priority queue the leaf representing the smallest deadline that is larger than the deadline of the new on-line task. We then construct for this on-line task a priority queue in the first layer of the priority-queue hierarchy of this leaf node. If there is already a priority queue in the first layer, we merge the new first-layer priority queue with the existing one and put the merged queue in the second layer. If there is a priority queue in the second layer, we merge the old one and the new one and put the merged queue in the third layer, and so on. The smallest value stored in the priority-queue hierarchy is the smallest value stored in all priority queues in this hierarchy. To update an reservation-array element or a group of consecutive elements stored in the priority-queue hierarchy, we need to update all priority queues in the hierarchy at the same time. This operation takes $O((\log n)^2)$ instead of $O(\log n)$. Since there are $O(n)$ updates, the total time required to update the reservation array is $O(n(\log n)^2)$. The run-time for constructing a priority-queue hierarchy is easy to calculate. When two priority queues of the ith layer are merged, the new priority queue is in the $(i+1)$th layer. Therefore, the reservation array element of each on-line task is stored in any layer of the hierarchy at most once. The number of layers in the hierarchy is at most $O(\log n)$; so the total time spent for constructing all priority-queue hierarchies is at most $O(n \log n)$. Therefore, the total run-time of the OAR algorithm is at most $O(n(\log n)^2)$.

5. On-line tasks ready upon arrival in the presence of off-line tasks. In §3, we showed that in Case 1, where there is no off-line task and on-line tasks are ready upon arrival, the reservation list can be updated in a simple way. This simple way to update the reservation list is used in the NORA algorithm. It remains correct if there is only one off-line task among on-line tasks that are ready upon arrival and the deadline of the off-line task is the latest among all deadlines. This observation points us to a way to simplify the OAR algorithm to handle Case 3, where on-line tasks are ready for execution when they arrive and there are off-line tasks with arbitrary ready times. We call this simplified algorithm the ORA (On-line task Ready upon Arrival) algorithm. Its time complexity is $O(n \log n)$.

Without loss of generality, let $T_1, T_2, \ldots, T_u$ be the u off-line tasks, and their deadlines are such that $0 < d_1 < d_2 < \cdots < d_u$. The idea behind the ORA algorithm is as follows: we divide the time into $u+1$ intervals, $I_1 = (0, d_1]$, $I_2 = (d_1, d_2], \ldots, I_{u+1} = (d_u, \infty]$. We maintain $u+1$ reservation lists $R_1, R_2, \ldots, R_{u+1}$. Before the processor begins executing any task, the reservation list $R_i (i \leq u)$ contains only the reserved interval $[d_i - m_i, d_i]$ for the off-line task T_i, and the reservation list R_{u+1} is empty. After the processor begins execution and on-line tasks arrive, we make the reservation for an on-line task whose deadline is in the interval I_i in the reservation list R_i. The simple operations for updating the reservation of on-line tasks used in the NORA algorithm are used to update each of the reservation lists.

In order to maintain and use the information provided by the $u+1$ reservation lists together, we also maintain a reservation array P as is done in the ORA algorithm. There is an element $P(i)$ for each reservation list R_i. As will become evident later, in deciding when mandatory tasks must be scheduled, the ORA algorithm, like the NORA algorithm, only checks whether the beginning of the earliest reserved interval in a reservation list is reached. For this reason, it suffices for us to keep track of the length l_i of time between the beginning of the earliest reserved interval and the end of the latest reserved interval in each of the reservation list R_i. The latter is the deadline d_i of the off-line task T_i for $i \leq u$ and is the latest deadline of on-line tasks whose deadlines are after d_u. We denote this deadline by d_{u+1}. Clearly, l_i is equal to the sum of the lengths of all reserved intervals and gaps between reserved intervals in the reservation list R_i.

Initially, before the processor begins execution, each of the reservation lists R_i for $i \leq u$ contains one reserved interval $[d_i - m_i, d_i]$ for the off-line task T_i, and R_{u+1} is empty. l_i is equal to m_i for $i \leq u$ and is equal to zero for $i = u+1$. The initial value of $P(i)$ is given by

$$P(i) = d_i - l_i - \sum_{d_j < d_i} m_j \tag{2}$$

for $i \leq u$, and $P(u+1) = d_n$. The operations of the ORA algorithm are similar to the operations of the OAR algorithm. $P(i)$'s are updated when any task becomes ready, when the mandatory task of the current task completes, when a new on-line task arrives, and when some deadline becomes a bottleneck. A difference between the two algorithms is in how $P(i)$'s are updated. During each update, we first update the reservation list R_i in which the reservation of the current task was made and compute the new value of l_i. Let σ_k be the processing time of the completed portion of the mandatory task M_k. The value of $P(i)$ is set to be

$$P(i) = d_i - l_i - \sum_{d_k < d_i} (m_k - \sigma_k). \tag{3}$$

Specifically, let $\Delta\sigma$ denote the processing time of the mandatory portion of the current task that is completed since the last update of the array P. To update the reservation of the task, we update the reservation list R_i by removing a reserved interval(s) of (total) length $\Delta\sigma$ from the beginning of R_i. Let $\Delta\sigma'$ be the difference between the lengths of the time from the beginning of the earliest reserved interval to the end of the latest reserved interval in R_i before and after R_i is updated. When a new task T_l arrives, we make a reservation for M_l in the reservation list R_i if the deadline of T_l is in the interval I_i. This reservation is done by reverse scheduling M_l in the gaps between the reserved intervals in R_i. Let Δl be the increase in the length of the time interval from beginning of the earliest reserved interval and the end of the latest reserved interval in R_i due to this new reservation. We subtract Δl from $P(i)$ and subtract m_l from all elements $P(k)$, where $d_k > d_i$.

We note that at any time t when $P(i) \leq t$, the beginning of the reservation list R_i is reached. This fact means that all the time from the current time to the beginning t' of the first gap in the reservation list R_i is reserved for mandatory tasks and the processor does not have time to execute any optional tasks in this interval. The ready mandatory tasks are scheduled on the EDF basis in the interval between the current time and t'.

We use a binary-tree priority queue, the same as the one used in the OAR algorithm, to store the $u+1$ reservation-array elements. This time we do not need a priority-queue hierarchy. The total run time for updating the reservation lists and the reservation array is $O(n \log n)$ because the reservation for each on-line task is made in one reservation list. Therefore, the run-time of the ORA algorithm is $O(n \log n)$.

6. Summary. In this paper, we examined the on-line scheduling problem in the imprecise computation model to minimize total error. There are four different cases to be considered. We proposed three algorithms that are solutions of these four cases. Two of these algorithms have time complexity $O(n \log n)$, and the other has time complexity $O(n \log^2 n)$.

The algorithms described here are greedy in that they try to schedule as much of each optional task as possible. We have shown that this strategy leads to schedules whose total errors are as small as possible. However, when trying to finish more optional tasks with earlier deadlines, the execution of some mandatory tasks with later deadlines may be postponed. This is the main cause of suboptimality of these algorithms in terms of the chance to produce feasible schedules of arbitrary task systems. They are optimal only when the given task system satisfies the feasible mandatory constraint. In this paper, we assumed that tasks have identical weights. A paper for the weighted version of this problem is in preparation.

REFERENCES

[1] K. J. Lin, S. Natarajan, W. S. Liu, and T. Krauskopf, *Concord: A system of imprecise computations*, in Proc. 1987 IEEE Compsac, IEEE Press, Piscataway, NJ, 1987, pp. 75–81.

[2] J. W. S. Liu, K. J. Lin, and S. Natarajan, *Scheduling real-time, periodic jobs using imprecise results*, in Proc. 8th IEEE Real-Time Systems Symposium, IEEE Press, Piscataway, NJ, 1987, pp. 210–217.

[3] J. Y. Chung and J. W. S. Liu, *Performance of algorithms for scheduling periodic jobs to minimize average error*, in Proc. 9th IEEE Real-Time Systems Symposium, IEEE Press, Piscataway, NJ, 1988, pp. 142–151.

[4] J. W. S. Liu, K. J. Lin, W. K. Shih, A. C. Yu, J. Y. Chung, and W. Zhao, *Algorithms for scheduling imprecise computations*, Comput. Magazine, 24 (1991), pp. 58–68.

[5] W. K. Shih, J. W. S. Liu, and J. Y. Chung, *Algorithms for scheduling imprecise computations with timing constraints*, in Proc. 10th IEEE Real-Time Systems Symposium, IEEE Press, Piscataway, NJ, 1989, pp. 12–19; SIAM J. Comput., 20 (1991), pp. 537–552.

[6] J. Blazewicz and G. Finke, *Minimizing mean weighted execution time loss on identical and uniform processors*, Inform. Process. Lett., 24 (1987), pp. 259–263.

[7] A. L. Liestman and R. H. Campbell, *A fault-tolerant scheduling problem*, IEEE Trans. Software Engrg., SE-12 (1986), pp. 1089–1095.

[8] E. K. P. Chong and W. Zhao, *Performance evaluation of scheduling algorithms for imprecise computer systems*, Technical Report, Department of Computer Science, University of Adelaide, Adelaide, SA, Australia, September 1988.

[9] ———, *User controlled optimization in task scheduling for imprecise computer systems*, Technical Report, Department of Computer Science, University of Adelaide, Adelaide, SA, Australia, October 1988.

[10] J. Y-T. Leung, T. W. Tam, C. S. Wong, and G. H. Young, *Minimizing mean flow time with error constraints*, in Proc. 10th IEEE Real-Time Systems Symposium, IEEE Press, Piscataway, NJ, 1989, pp. 1–11.

[11] E. L. Lawler and J. M. Moore, *A functional equation and its application to resource allocation and scheduling problem*, Management Sci., 16 (1969), pp. 77–84.

[12] S. Baruah, G. Koren, D. Mao, B. Mishra, A. Raghunathan, L. Rosier, D. Shasha, and F. Wang, *On the competitiveness of on-line real-time task scheduling*, in Proc. 12th IEEE Real-Time Systems Symposium, IEEE Press, Piscataway, NJ, 1991, pp. 106–115.

SIAM J. COMPUT.
Vol. 25, No. 6, pp. 1123–1143, December 1996

001

KOLMOGOROV COMPLEXITY AND INSTANCE COMPLEXITY OF RECURSIVELY ENUMERABLE SETS*

MARTIN KUMMER[†]

Abstract. The way in which way Kolmogorov complexity and instance complexity affect properties of recursively enumerable (r.e.) sets is studied. The well-known $2\log n$ upper bound on the Kolmogorov complexity of initial segments of r.e. sets is shown to be optimal, and the Turing degrees of r.e. sets which attain this bound are characterized. The main part of the paper is concerned with instance complexity, introduced by Ko, Orponen, Schöning, and Watanabe in 1986, as a measure of the complexity of individual instances of a decision problem. They conjectured that for every r.e. nonrecursive set, the instance complexity is infinitely often at least as high as the Kolmogorov complexity. The conjecture is refuted by constructing an r.e. nonrecursive set with instance complexity logarithmic in the Kolmogorov complexity. This bound is optimal up to an additive constant. In the other extreme, the conjecture is established for many classes of complete sets, such as weak-truth-table-complete (wtt-complete) and Q-complete sets. However, there is a Turing-complete set for which it fails.

Key words. Kolmogorov complexity, instance complexity, recursively enumerable sets, complete sets

AMS subject classifications. 03D15, 03D32, 68Q15

1. Introduction.

In recent years, ideas from Kolmogorov complexity have turned out to be of central importance for both foundations and applications, as witnessed by the textbooks of Calude [3] and Li and Vitányi [10].

Intuitively, Kolmogorov complexity measures the "descriptional complexity" of a string x. It is defined as the length of the shortest program that computes x from the empty input. Accordingly, the Kolmogorov complexity of initial segments of a set A is considered as a measure of the "randomness" of A. It is well known that for recursively enumerable (r.e.) sets, the Kolmogorov complexity of initial segments of length n is bounded by $2\log n$. We show that this bound is optimal and characterize the Turing degrees of r.e. sets which attain this bound as the array nonrecursive degrees (a proper subclass of the r.e. nonrecursive degrees), introduced by Downey, Jockusch, and Stob [5]. This characterization yields a curious "gap phenomenon": in every r.e. Turing degree, either we find an r.e. set which attains the $2\log n$ bound or, for all of its r.e. sets and every $\epsilon > 0$, the Kolmogorov complexity is almost always less than $(1+\epsilon)\log n$. This bound is tight because, by a result of Meyer and Chaitin, if the Kolmogorov complexity is always less than $\log n + O(1)$, then the set is recursive.

Ko, Orponen, Schöning, and Watanabe [9, 14] have recently introduced the notion of *instance complexity* as a measure of the complexity of individual instances of A. Informally, $\mathrm{ic}(x : A)$, the instance complexity of x w.r.t. A, is the length of the shortest total program which correctly computes $\chi_A(x)$ and does not make any mistakes on other inputs but is permitted to output "don't know" answers. It is easy to see that the Kolmogorov complexity of x is an upper bound for the instance complexity of x (up to a constant). A set A has *hard instances* if for infinitely many x's the instance

* Received by the editors June 1, 1994; accepted for publication (in revised form) February 2, 1995. A preliminary version of this paper appeared under the title "The instance complexity conjecture" in *Proc. 10th Annual Conference on Structure in Complexity Theory,* IEEE Computer Society Press, Los Alamitos, CA, 1995, pp. 111–124.

[†] Institut für Logik, Komplexität, und Deduktionssysteme, Universität Karlsruhe, D-76128 Karlsruhe, Germany (kummer@ira.uka.de).

complexity of x w.r.t. A is at least as high as the Kolmogorov complexity of x (up to a constant which may depend on A), i.e., the trivial upper bound is already optimal.

Orponen et al. conjectured in [13, 14] that every r.e. nonrecursive set has hard instances (the instance-complexity conjecture (ICC)). Buhrman and Orponen [2] proved ICC for m-complete sets. Tromp [16] proved that the instance complexity of x w.r.t. any nonrecursive set A is infinitely often at least logarithmic in the Kolmogorov complexity of x, but this lower bound was believed to be far from optimal.

However, we show that there is indeed an r.e. nonrecursive sets A with $\mathrm{ic}(x : A) \leq \log C(x) + O(1)$ for all x. This also establishes the first counterexample to the ICC and shows that the border between the instance complexity of recursive and nonrecursive sets is at $\log C(x)$ instead of $C(x)$. Since our construction places a lot of restrictions on A, the question arises of whether the ICC still holds for all complete sets even w.r.t. weak reducibilities.

We answer this positively by showing that the ICC holds for all weak-truth-table-complete (wtt-complete) sets and all Q-complete sets, i.e. it holds for all reducibilities stronger than Turing reducibility. However, we show that it fails for some Turing-complete set. In fact, we construct an effectively simple set for which it fails. (It is known that every effectively simple set is Turing complete.) Since every strongly effectively simple set is Q-complete, it follows that the ICC holds for all strongly effectively simple sets but not for all effectively simple sets, i.e., we have obtained a very close separation. In addition, we show that the ICC holds for all hyperhypersimple sets (which are known to be neither Q-complete nor wtt-complete).

We also investigate a weak version of instance complexity, where programs may not halt instead of giving "don't know" answers. We show that this yields a much stronger notion of "hard instances," e.g., while all c-complete sets still have hard instances in this stronger sense, this no longer holds for d-complete sets.

The resource-bounded version of instance complexity is also well studied; we refer the reader to [2, 7, 8, 14]. It seems that in this setting, the (suitably adapted) ICC is likely to hold, e.g., in [7], the ICC is proved w.r.t. polynomial-space-bounded computations; however, it is also shown that the polynomial-time-bounded version of the ICC is oracle dependent.

2. Notation and definitions.

Our notation generally follows that of Li and Vitányi [10]. For $p \in \{0,1\}^*$, $l(p)$ denotes the length of p; ϵ is the empty string. We write $\log(x)$ for $\log_2(x)$. We use the special symbol $\perp$ to denote the "don't know" output. χ_A is the characteristic function of A. We identify $\mathcal{N}$ and $\{0,1\}^*$ via the canonical correspondence as in [10, p. 11]. For the convenience of the reader, we recall some recursion-theoretic notions that are used later; for background, see the textbooks of Odifreddi [12] and Soare [15].

φ_i is the ith function in a standard enumeration of all partial recursive functions of one argument. If ψ is partial recursive, then $\psi_s(x)$ denotes the result, if any, of performing s steps in the computation of $\psi(x)$. $f(x)\downarrow$ means that f is defined on x and $f(x)\uparrow$ means that f is undefined on x. $\mathrm{dom}(f) = \{x : f(x)\downarrow\}$ and $\mathrm{range}(f) = \{f(x) : x \in \mathrm{dom}(f)\}$. $W_i = \mathrm{dom}(\varphi_i)$ is the ith r.e. set. $K = \{e : \varphi_e(e)\downarrow\}$ is the halting problem. If A is an r.e. set, then A_s denotes the set of elements enumerated into A before step s in some fixed recursive enumeration of A. D_n denotes the nth finite set in a canonical enumeration of all finite sets.

In addition to the well-known m-reducibility ($\leq_m$), truth-table reducibility ($\leq_{tt}$), and Turing reducibility ($\leq_T$) we also consider the following. *Q-reducibility:* $A \leq_Q B$ if there is a recursive function f such that $x \in A \Leftrightarrow W_{f(x)} \subseteq B$. *wtt-reducibility:*

$A \leq_{wtt} B$ if there is a recursive function g and a Turing reduction Φ such that $A = \Phi^B$ and in the computation of Φ^B, on input x, all queries are less than $g(x)$. *Positive reducibility:* $A \leq_p B$ if there is a recursive function f such that $x \in A \Leftrightarrow (\exists y)[y \in D_{f(x)} \wedge D_y \subseteq B]$. *Disjunctive reducibility:* $A \leq_d B$ if there is a recursive function f such that $x \in A \Leftrightarrow D_{f(x)} \cap B \neq \emptyset$. *Conjunctive reducibility:* $A \leq_c B$ if there is a recursive function f such that $x \in A \Leftrightarrow D_{f(x)} \subseteq B$.

A sequence $\{U_i\}_{i \in \mathcal{N}}$ of finite sets is a *strong array* if there is a recursive function f such that $U_i = D_{f(i)}$. A sequence $\{U_i\}_{i \in \mathcal{N}}$ of finite sets is a *weak array* if there is a recursive function f such that $U_i = W_{f(i)}$. An array is *disjoint* if its members are pairwise disjoint. An r.e. set A is *hypersimple* if there is no disjoint strong array $\{U_i\}_{i \in \mathcal{N}}$ such that $\overline{A} \cap U_i \neq \emptyset$ for all i. An r.e. set A is *hyperhypersimple* if there is no disjoint weak array $\{U_i\}_{i \in \mathcal{N}}$ such that $\overline{A} \cap U_i \neq \emptyset$ for all i.

After these preliminaries, we now turn to the central definitions of Kolmogorov complexity and instance complexity.

DEFINITION 2.1 (Chaitin, Kolmogorov, Solomonoff). *For any partial recursive mapping* $U : \{0,1\}^* \times \{0,1\}^* \rightarrow \{0,1\}^* \cup \{\perp\}$ *and any* $x \in \{0,1\}^*$, *we define* $C_U(x) = \min\{l(p) : U(p, \epsilon) = x\}$, *the* Kolmogorov complexity *of* x *in* U. *If no such* p *exists, then* $C_U(x) = \infty$.

It is helpful to think of U as an *interpreter* which takes a *program* p and an input z and produces the output $U(p, z)$

Instance complexity was introduced in [9] in order to study the complexity of single instances of a decision problem.

DEFINITION 2.2 ((Ko, Orponen, Schöning, Watanabe 1986)). *Let* $A \subseteq \{0,1\}^*$. *A function* $f : \{0,1\}^* \rightarrow \{0,1,\perp\}$ *is called* A-consistent *if* $[f(x) = \chi_A(x) \vee f(x) = \perp$ *for all* $x \in \text{dom}(f)]$. *The* instance complexity *of* x *w.r.t.* A *in* U *is defined as* $\text{ic}_U(x : A) = \min\{l(p) : \lambda z. U(p, z)$ *is a* total *A-consistent function such that* $U(p, x) = \chi_A(x)\}$. *If no such* p *exists, then* $\text{ic}_U(x : A) = \infty$.

If we drop the requirement that $\lambda z.\ U(p, z)$ is total in the definition of ic_U, then we obtain a weaker notion of instance complexity, which we denote by $\overline{\text{ic}}_U(x : A)$. Note that $\overline{\text{ic}}_U(x : A) \leq \text{ic}_U(x : A)$ for all x and A.

It is well known (see [10]) that there exist "optimal" partial recursive functions U such that for every partial recursive mapping U', there is a constant c with $C_U(x) \leq C_{U'}(x) + c$, $\text{ic}_U(x : A) \leq \text{ic}_{U'}(x : A) + c$, and $\overline{\text{ic}}_U(x : A) \leq \overline{\text{ic}}_{U'}(x : A) + c$ for all x and A.

For the following, we fix an optimal mapping U and write $C(x)$, $\text{ic}(x : A)$, and $\overline{\text{ic}}(x : A)$ for $C_U(x)$, $\text{ic}_U(x : A)$, and $\overline{\text{ic}}_U(x : A)$, respectively. We also write $U_s(p, z)$ for the result, if any, after s steps of computation of U with input (p, z). $C^s(x)$ denotes the approximation to $C(x)$ after s steps of computation (i.e., with U_s in place of U in the definition of $C(x)$). Clearly, $C^{s+1}(x) \leq C^s(x)$ and $C^t(x) = C(x)$ for all sufficiently large t.

The instance complexity of x can be bounded by the Kolmogorov complexity of x in the sense that for every set A, there is a constant c such that $\text{ic}(x : A) \leq C(x) + c$ for all x. Informally, x is a hard instance of A if this upper bound is also a lower bound. This was the motivation for the following definition (which is independent of the choice of the optimal U).

DEFINITION 2.3 (Ko, Orponen, Schöning, Watanabe 1986). *A set* A *has* hard instances *if there is a constant* c *such that*

$$\text{ic}(x : A) \geq C(x) - c \qquad \textit{for infinitely many } x\text{'s}.$$

If the condition holds with $\overline{\text{ic}}$ *in place of* ic, *we say that* A *has hard instances* w.r.t. $\overline{\text{ic}}$.

Remark. The difference between ic and $\overline{\text{ic}}$ is perhaps best explained by an example. Suppose that A is an r.e. set and we want to define a program p such that it witnesses $\text{ic}(x : A) \leq |p|$ for all x with $C(x) < n$. Since p has to be total, we have to define it for every input z at some step s. If z already appears in A_s, there is no problem; we set $U(p, z) = 1$. If z has not yet appeared and $C^s(z) \geq n$, we could try to define $U(p, z) = \perp$, but this can later become incorrect if it turns out that $C(z) < n$. If we set $U(p, z) = 0$ and z later appears in A, then p is also incorrect.

In the case of $\overline{\text{ic}}$, we have more freedom. We may leave $U(p, z)$ undefined until $C^s(z) < n$ at some stage s. If this never happens, then $U(p, z)$ is undefined and $C(z) \geq n$, which is fine. Still, the second source of error remains. If $C^s(z) < n$ and z has not yet appeared in A at stage s, we have to define $U(p, z)$, and the best we can do is to set $U(p, z) = 0$. However, this may later turn out to be incorrect.

3. A version of Barzdin's lemma. In this section, we consider the Kolmogorov complexity of initial segments of r.e. sets. For $A \subseteq \mathcal{N}$ and $n \in \mathcal{N}$, we write $\chi_A \restriction n$ for the string $\chi_A(0) \dots \chi_A(n)$

Let us first recall what was previously known. The uniform complexity of a string $\sigma = b_1 \dots b_n$ of length n is defined as

$$C(\sigma; n) = \min\{l(p) : U(p, m) = b_1 \dots b_m \text{ for all } m \leq n\}.$$

We write $C(\chi_A; n)$ for $C(\chi_A \restriction (n-1); n)$. Barzdin (see [1]; see also [10, Thm. 2.18]) characterized the worst case of the uniform complexity of r.e. sets:

(i) For every r.e. set A, there is a constant c such that for all n, $C(\chi_A; n) \leq \log n + c$.

(ii) There is an r.e. set A such that $C(\chi_A; n) \geq \log n$ for all n.

Let us now look at the standard Kolmogorov complexity $C(\chi_A \restriction n)$. Utilizing a result of Meyer [11, p. 525], Chaitin proved that if there is constant c such that for all n, $C(\chi_A \restriction n) \leq \log n + c$, then A is recursive [4, Thm. 6], [10, Exercise 2.43].

For every r.e. set A, there is a constant c such that $C(\chi_A \restriction n) \leq 2 \log n + c$ for all n (see [10, Exercise 2.59]). On the other hand, there is no r.e. set A such that $C(\chi_A \restriction n) \geq 2 \log n - O(1)$ for all n. This follows from the argument in [10, Exercise 2.58].

In [10, Exercise 2.59], it is stated as an open question (attributed to Solovay) whether the upper bound $2 \log n$ is optimal. The following result shows that this is indeed the case. For ease of conversation, we say that A is *complex* if there is a constant c such that $C(\chi_A \restriction n) \geq 2 \log n - c$ for infinitely many $n \in \mathcal{N}$.

THEOREM 3.1. *There is an r.e. complex set.*

Proof. Let $t_0 = 0, t_{k+1} = 2^{t_k}$, and $I_k = (t_k, t_{k+1}]$ for all $k \geq 0$. (I_k) is a sequence of exponentially increasing half-open intervals.

Let $f(k) = \sum_{i=t_k+1}^{t_{k+1}} (i - t_k + 1)$ and $g(k) = \max\{l : 2^{l+1} - 1 < f(k)\}$. Note that $f(k) = \frac{1}{2} t_{k+1}^2 (1 - o(1))$ and $g(k) = 2 \log t_{k+1} - 2 - o(1)$, for $k \to \infty$. We enumerate an r.e. set A in steps as follows.

Construction:

Step 0: Let $A_0 = \emptyset$.

Step $s+1$: Let $A_{s+1} = A_s$. For $k = 0, \dots, s$ do the following: if $C^s(\chi_{A_s} \restriction n) \leq g(k)$ for all $n \in I_k$, then enumerate $\min(\overline{A}_s \cap I_k)$ into A_{s+1}.

End of Construction.

Let $A = \cup_{s \geq 0} A_s$. Suppose for the sake of contradiction that $C(\chi_A \restriction n) \leq g(k)$ for all $n \in I_k$. Then we eventually enumerate every $n \in I_k$ into A. Note that for fixed n, there are at least $n - t_k + 1$ different strings $\sigma = \chi_{A_s} \restriction n$ with $l(\sigma) = n+1$ and $C(\sigma) \leq g(k)$. (The suffix of $\chi_{A_s} \restriction n$ runs through $1^x 0^{n-t_k-x}$ for $x = 0, \ldots, n - t_k$.) Thus there are at least $f(k)$ many different strings which all have Kolmogorov complexity at most $g(k)$. This contradicts the fact that $f(k) > 2^{g(k)+1} - 1$.

Therefore, for every k, there exists $n \in I_k$ with $C(\chi_A \restriction n) > g(k)$, i.e., $C(\chi_A \restriction n) > g(k) \geq 2 \log n - 2 - o(1)$. Thus A is complex. □

We now characterize the T-degrees of r.e. complex sets. Downey, Jockusch, and Stob [5] introduced the notion of an *array nonrecursive* set. This captures precisely those r.e. sets that arise in multiple-permitting arguments. In [5, 6], several other natural characterizations of this degree class are given.

An r.e. set A is called *array nonrecursive* w.r.t. $\{F_k\}_{k \in \mathcal{N}}$ if

$$(\forall e)(\exists^\infty k)[W_e \cap F_k = A \cap F_k].$$

Here $\{F_k\}_{k \in \mathcal{N}}$ denotes a very strong array. This means that $\{F_k\}_{k \in \mathcal{N}}$ is a strong array of pairwise-disjoint sets which partition $\mathcal{N}$ and satisfy$|F_k| < |F_{k+1}|$ for all $k \in \mathcal{N}$.

An r.e. set is *array nonrecursive* if it is *array nonrecursive* w.r.t. some very strong array $\{F_k\}_{k \in \mathcal{N}}$. A T-degree is called *array nonrecursive* if it contains an r.e. array nonrecursive set. Not every r.e. nonrecursive T-degree is array nonrecursive [5, Thm. 2.10].

THEOREM 3.2. *The T-degrees containing an r.e. complex set coincide with the array nonrecursive T-degrees. In addition, if A is r.e. and not of array nonrecursive T-degree, then for every unbounded, nondecreasing, total recursive function f, there is a constant c such that*

$$C(\chi_A \restriction n) \leq \log n + f(n) + c \qquad \textit{for all } n \in \mathcal{N}.$$

Proof. Note that in order to make A complex, it suffices to perform the construction from Theorem 3.1 for infinitely many intervals. Suppose that A is array nonrecursive w.r.t. $\{I_k\}_{k \in \mathcal{N}}$, and fix an enumeration $\{A_s\}_{s \in \mathcal{N}}$ of A. We modify the construction of Theorem 3.1 and enumerate an auxiliary r.e. set B as follows. If in step $s+1$, $C^s(\chi_{A_s} \restriction n) \leq g(k)$ and $B_s \cap I_k = A_s \cap I_k$, then enumerate $\min(\overline{A}_s \cap I_k)$ into B_{s+1}.

By hypothesis on A, there are infinitely many k's such that $A \cap I_k = B \cap I_k$. By the proof of Theorem 3.1, this implies that A is complex. In [5, Thm. 2.5], it is shown that every array nonrecursive T-degree contains a set A which is array nonrecursive w.r.t. $\{I_k\}_{k \in \mathcal{N}}$; thus it contains an r.e. complex set.

For the converse, we use [5, Thm. 4.1]. It states that if A is r.e. and is not of array nonrecursive T-degree, then for every total function $g \leq_T A$, there is a total recursive approximation $\overline{g}(x,s)$ such that $\lim_s \overline{g}(x,s) = g(x)$ and $|\{s : \overline{g}(x,s) \neq \overline{g}(x,s+1)\}| \leq x$ for all $x \in \mathcal{N}$. Informally, $g(x)$ can be recursively approximated with at most x mind changes. In [5], this is stated only for 0/1-valued g, but their proof provides the more general version.

Let A be r.e. and not of array nonrecursive T-degree. Assume that we are given any total recursive, nondecreasing, unbounded function f. Let $m(x) = 1 + \max\{n : f(n) \leq x\}$; m is total recursive. Let $g(x) = \chi_A \restriction m(x)$. Since g is recursive in A, there is a total recursive approximation $\overline{g}(x,s)$ as above.

How can we describe $\chi_A \restriction n$? Given n, we compute $n' = \min\{x : m(x) > n\}$. Then we simulate $\overline{g}(n',s)$ until it outputs $g(n')$, which gives us $\chi_A \restriction n$. In order to perform the simulation, we only need to know the exact number $k \leq n'$ of mind changes of $\overline{g}(n', s)$. Thus $\chi_A \upharpoonright n$ is specified by the pair $\langle k, n \rangle$, which can be encoded by a string of length $\log n + 2\log(k+1) + O(1)$. Since $m(k-1) \leq n$, we have $f(n) \geq k$ by the definition of m. Thus we get

$$C(\chi_A \upharpoonright n) \leq \log n + 2\log(k+1) + O(1) \leq \log n + f(n) + O(1).$$

This completes the proof of the theorem. □

Note that Theorem 3.2 entails the following curious gap phenomenon. For every r.e. T-degree **a**, there are only two cases:

1. There is an r.e. set $A \in \mathbf{a}$ such that $(\exists^\infty n)[C(\chi_A \upharpoonright n) \geq 2\log n - O(1)]$.
2. There is no r.e. set $A \in \mathbf{a}$ and $\epsilon > 0$ such that $(\exists^\infty n)[C(\chi_A \upharpoonright n) \geq (1+\epsilon)\log n - O(1)]$.

4. The ICC fails. In this section, we determine the least possible instance complexity of r.e. nonrecursive sets. Here it is convenient to take A as a subset of $\{0,1\}^*$. Clearly, if A is recursive, then $\mathrm{ic}(x : A)$ is bounded by a constant for all x. The next result (another gap theorem) shows that for infinitely many x, $\mathrm{ic}(x : A)$ must be at least logarithmic in $C(x)$ if A is nonrecursive.[1]

THEOREM 4.1. *If* $\mathrm{ic}(x : A) \leq \log C(x) - 1$ *for almost all* x, *then* A *is recursive.*

Proof. Let $P_k = \{0,1\}^{\leq k}$ and let $\mathcal{P}(P_k)$ denote the set of all subsets of P_k. Uniformly in k, we enumerate a finite set $B_k \subseteq \{0,1\}^*$.

Construction:

Step 0: Let $S_k = \mathcal{P}(P_k)$ and $B_k = \emptyset$.

Step $n+1$: Search via dovetailing for $I \in S_k$, $x \in \{0,1\}^*$, and $s \in \mathcal{N}$ such that $U_s(p, x) = \perp$ for all $p \in I$. If such an I is found, then enumerate x into B_k, remove I from S_k, and go to step $n+2$.

End of Construction.

Note that B_k is nonempty since $I = \emptyset$ trivially satisfies the condition for all x and s. Also, at most $|\mathcal{P}(P_k)| = 2^{|P_k|}$ elements are enumerated into B_k and B_k is uniformly r.e. Thus there is a partial recursive function $\psi : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^*$ such that $\psi(\{0,1\}^{|P_k|}, \epsilon) = B_k$ for all k. In particular, $C_\psi(x) \leq |P_k| = 2^{k+1} - 1$ for all $x \in B_k$. Choose a constant c such that $C(x) \leq C_\psi(x) + c$ for all x.

Let $A \subseteq \{0,1\}^*$ be given and suppose that $\mathrm{ic}(x : A) \leq \log C(x) - 1$ for almost all x. Then $\mathrm{ic}(x : A) \leq \log(C_\psi(x) + c) - 1$, so $\mathrm{ic}(x : A) \leq \log C_\psi(x)$ for almost all x. Since $\lfloor \log C_\psi(x) \rfloor \leq \lfloor \log(2^{k+1} - 1) \rfloor = k$ for all $x \in B_k$, we can choose k large enough such that for all $x \in B_k$, we have $\mathrm{ic}(x : A) \leq k$.

Thus for each $x \in B_k$, there is $p \in P_k$ such that $\lambda z.\ U(p, z)$ is a total A-consistent function with $U(p, x) = \chi_A(x)$. Let $I_0 = \{p \in P_k : \lambda z.\ U(p, z)$ is a total A-consistent function$\}$. This set is nonempty since B_k is nonempty.

Now consider the construction of B_k. Note that I_0 cannot be removed from S_k. Otherwise, there exists $x \in B_k$ such that $C_\psi(x) \leq 2^{k+1} - 1$ and $U(p, x) = \perp$ for all $p \in I_0$, i.e., $\mathrm{ic}(x : A) > k$, contradicting the choice of k. Since I_0 is never removed from S_k, it follows from the construction of B_k that for every x, there is $p \in I_0$ with $U(p, x) = \chi_A(x)$. Thus if we amalgamate all of the functions $U(p, -)$ with $p \in I_0$, we get a recursive characteristic function of A, i.e., A is recursive. □

We prove that the lower bound of Theorem 4.1 is tight even for r.e. nonrecursive sets. This refutes the ICC of Orponen et al. [13, 14], [10, Exercise 7.41], which states

[1] This result was previously obtained by Tromp [16].

that every r.e. nonrecursive set has hard instances. In contrast, our result together with Theorem 4.1 shows that the true threshold between the instance complexity of recursive and nonrecursive sets is $\log C(x)$ instead of $C(x)$.

THEOREM 4.2. *There is an r.e. nonrecursive set A and a constant c such that*

$$\mathrm{ic}(x : A) \leq \log C(x) + c \quad \textit{for all } x.$$

Proof. It suffices to construct an r.e. nonrecursive set A and a partial recursive function ψ such that $\mathrm{ic}_\psi(x : A) \leq \log C(x) + 2$ for almost all x. In the following, we write ψ_p for $\lambda z.\, \psi(p, z)$.

Let $E_k = \{x : C(x) < 2^k - 2\}$ for $k \geq 1$. We want to establish that $\mathrm{ic}_\psi(x : A) \leq k$ for all $x \in E_k$. Let $M_k = \{p_{k,1}, \ldots, p_{k,2^k-2}\}$ denote the set of the first $2^k - 2$ strings of length k. The idea is that every $\psi_{p_{k,i}}$ is A-consistent and for each $x \in E_k$ there is $p \in M_k$ such that ψ_p witnesses that $\mathrm{ic}_\psi(x : A) \leq k$. There is, however, some difficulty in combining this with the requirement of making A nonrecursive.

The basic idea to satisfy the latter requirement is as follows. For each $e \geq 1$, we establish a unique diagonalization value d_e and then wait until d_e is enumerated into W_e; if this ever happens, we enumerate d_e into A. Here we interpret W_e as a set of strings. Hence this strategy makes sure that $\overline{A}$ is not r.e., so A is an r.e. nonrecursive set.

Suppose that d_e appears in E_k before it appears in W_e. If we define $\psi_{p_{k,i}}(d_e) = 0$ for some i, then since $\psi_{p_{k,i}}$ should be A-consistent, we can no longer enumerate d_e into A. This threatens our diagonalization strategy. On the other hand, we should certainly make sure that $\mathrm{ic}_\psi(d_e : A) \leq k$.

This conflict is solved by a finite-injury priority argument:

If $e \geq k$ and we are forced to define $\psi_{p_{k,i}}(d_e) = 0$, then we assign a new, much larger value to d_e and try to diagonalize at this new value. Note that d_e is changed only finitely often because there are only finitely many values which may appear in E_k for some $k \leq e$. Thus the value of d_e eventually stabilizes and the eth diagonalization strategy goes through with this final value.

If $e < k$, then we do not use $\psi_{p_{k,i}}$ to ensure that $\mathrm{ic}_\psi(d_e : A) \leq k$. Thus we define $\psi_{p_k,i}(d_e) = \perp$, which certainly maintains the A-consistency. Instead, we will have two special programs $\tau_{e,1}$ and $\tau_{e,2}$ of length e (which are not in M_e; this is the reason why we have left out two strings) to witness that $\mathrm{ic}_\psi(d_e : A) \leq e < k$. More precisely, if the final d_e-value is not enumerated into A, then $\psi_{\tau_{e,1}}$ will be the correct function. If the final d_e-value is enumerated into A, then $\psi_{\tau_{e,1}}$ will not be A-consistent, but $\psi_{\tau_{e,2}}$ is used as a backup function.

It remains to explain how only $|M_k|$ many programs can take care of all of the elements in E_k, which may be up to $2^{|M_k|} - 1$ many. We show in an example how two programs p_1 and p_2 can take care of $3 = 2^2 - 1$ elements. (For simplicity, we drop the distinction between numbers and strings.) At the beginning, ψ_{p_1} and ψ_{p_2} are undefined. Now in step s_1, the first element $x_1 < s_1$ appears. We let $\psi_{p_1}(x) = \chi_A(x)$ for all $x \leq s_1$. In the following steps s, we define $\psi_{p_1}(s) = \perp$ until the second element x_2 appears, say at step $s_2 > x_2$. If $x_2 \leq s_1$, we do nothing. If $x_2 > s_1$, then we define $\psi_{p_2}(x) = \chi_A(x)$ for all $x \leq s_2$, and in the following steps t, we define $\psi_{p_2}(t) = \perp$. The point is that ψ_{p_2} also takes care of x_1; thus we suspend the definition of ψ_{p_1} until a third element x_3 appears at step $s_3 > x_3$. If $x_3 > s_2$, then we resume the definition of ψ_{p_1} and let $\psi_{p_1}(x) = \chi_A(x)$ for all $s_2 < x \leq s_3$. For arguments $t > s_3$, we define both functions to be equal to $\perp$. Note that now p_1 and p_2 together take care of x_1, x_2, and x_3.

This idea is easily generalized. Let $\text{succ}(\sigma)$ denote the lexicographical successor of σ, i.e., if $\sigma = b_1 \dots b_n \neq 1^n$, then $\text{succ}(\sigma) = 0^{i-1}1b_{i+1}\dots b_n$, where $i = \min\{j : b_j = 0\}$. For example, $\text{succ}^{(m)}(000)$ for $m = 0, \dots, 7$ yields the sequence 000, 100, 010, 110, 001, 101, 011, 111. The programs $p_{k,i} \in M_k$ with $\text{succ}^{(m)}(0^{|M_k|})(i) = 1$ take care if exactly m elements are enumerated into E_k. For example, the first three elements are handled by $p_{k,1}$ and $p_{k,2}$, the first four by $p_{k,3}$, the first five by $p_{k,1}$ and $p_{k,3}$, etc. (In the implementation below, we count only those elements which are not d_e-values for some $e < k$.) Note that since $m \leq 2^{|M_k|} - 1$, succ is never applied to $1^{|M_k|}$.

We now turn to the detailed implementation. First, we fix some additional notation and conventions. Let $\langle -, - \rangle$ denote a recursive pairing function which is increasing in its second argument. We assume that elements of E_k are enumerated in steps such that in each step, at most one new element is enumerated; also, if x is enumerated in step s, then $l(x) < s$. $W_{e,s}$ is the finite set of strings which are enumerated into W_e in at most s steps of computation.

In the construction, the variables e, i, j, k, n, s, and t denote numbers and p, x, and z denote strings. In addition, the following variables are used: $\psi_{p,s}$ is the finite portion of ψ_p constructed up to stage s; the ith bit of $\sigma_{k,s} \in \{0,1\}^{|M_k|}$ tells us whether $\psi_{p_{k,i}}$ is currently assigned to take care of the elements in E_k; $\text{len}(k,s)$ is the greatest length n such that our setup at stage s guarantees that $\text{ic}_\psi(x : A) \leq \log C^s(x) + 2$ for all $x \in E_{k,s}$ with $l(x) < n$; $d_e(s)$ is the current value of the eth diagonalization point. We call e "active" as long as no d_e-value has been enumerated into A; otherwise, we call e "passive." Therefore, if e is "passive," then we know that we have explicitly satisfied the eth diagonalization requirement. A_s denotes the finite set of elements which have been enumerated into A up to stage s.

Let $R(k,s) = \{d_e(s') : e < k \ \wedge \ s' \leq s\}$. As explained above, the programs in M_k do not need to take care of the elements in $R(k,s)$.

If one of the variables $v(s)$ is not explicitly changed at stage $s+1$, then we assume without further mention that $v(s+1) = v(s)$.

We first describe the construction of ψ_p for $p \in M_k$, $k \geq 1$. Then we define ψ_p for the two special values $p = \tau_{e,1}, \tau_{e,2}$ of each length e.

Construction:

Stage 0: Let $\psi_p = \lambda x.\uparrow$ for all $p \in \{0,1\}^*$. For all $k \geq 1$: $\sigma_{k,0} = 0^{|M_k|}$; $\text{len}(k,0) = 0$; $d_k(0) = 0^{\langle k,0 \rangle}$; declare k to be "active." Let $A_0 = \emptyset$.

Stage $s+1$:

Case I: s is even.

For $e = 0, \dots, s$, if e is active and $d_e(s) \in W_{e,s} - A_s$, then enumerate $d_e(s)$ into A and declare e "passive."

Case II: s is odd, $s = 2\langle k,t \rangle + 1$.

Let $\psi_{p_{k,i}}(x) = \perp$ for all i with $\sigma_{k,s}(i) = 1$ and all x with $l(x) = t$.

If a new element x, $l(x) < t$, enters E_k after exactly t steps, then act according to the following cases:

(a) If $x \in R(k,s)$ or $l(x) < \text{len}(k,s)$, then go to stage $s+2$.

(b) Otherwise, do the following:

Let $\sigma_{k,s+1} = \text{succ}(\sigma_{k,s})$ and $i = \min\{j : \sigma_{k,s+1}(j) = 1\}$. (At most $2^{|M_k|} - 1$ elements are enumerated in E_k, so we get $\sigma_{k,s+1} \in \{0,1\}^{|M_k|} - \{0^{|M_k|}\}$.)

Let $n = \min\{l(z) : z \notin \text{dom}(\psi_{p_{k,i}})\}$.

Define $\psi_{p_{k,i}}(z) = \chi_{A_s}(z)$ for all $z \notin R(k,s)$ such that $n \leq l(z) \leq t$.

Let $\psi_{p_{k,i}}(z) = \perp$, for all $z \in R(k,s)$ such that $n \leq l(z) \leq t$.

Let $\text{len}(k, s+1) = t+1$. For all active $e \geq k$, let $d_e(s+1) = 0^{\langle e,s+1\rangle}$.

Go to stage $s+2$.

End of Construction.

For each $e \geq 1$, we define

$$\psi_{\tau_{e,1}}(x) = \begin{cases} 0 & \text{if } x \in \text{range}(d_e); \\ \perp & \text{otherwise.} \end{cases}$$

If e is active at all stages, then let $\psi_{\tau_{e,2}} = \lambda x.\uparrow$. Otherwise, let s_e be the (unique) stage where e is declared "passive" and let

$$\psi_{\tau_{e,2}}(x) = \begin{cases} 0 & \text{if } (\exists t < s_e)[x = d_e(t) \neq d_e(s_e)]; \\ 1, & x = d_e(s_e); \\ \perp & \text{otherwise.} \end{cases}$$

Verification. Most of the following claims are standard. The crucial one is Claim 3(b–c).

CLAIM 1. *For all $e \geq 1$, we have the following:*

(a) *$l(d_e)$ is nondecreasing, and for all s, if $d_e(s) \neq d_e(s+1)$, then $l(d_e(s+1)) > s$.*

(b) $\text{range}(d_e)$ *is a uniformly recursive finite set*; $\text{range}(d_e) \cap \text{range}(d_{e'}) = \emptyset$ *for all $e' \neq e$.*

(c) *If $A \cap \text{range}(d_e)$ contains an element x, then $x = \lim_{s\to\infty} d_e(s)$.*

(d) *For all x and s, if $l(x) \leq s$ and, for all e, $x \neq d_e(s)$, then $x \neq d_e(s')$ for all e and all $s' \geq s$.*

(e) *A is r.e. and nonrecursive.*

Proof. (a) If $d_e(s) \neq d_e(s+1)$, then for some $s' \leq s$, $l(d_e(s)) = \langle e, s'\rangle < \langle e, s+1\rangle = l(d_e(s+1))$. Note that $\langle e, s+1\rangle > s$ since $\langle -,-\rangle$ is monotone in the second argument.

(b) It follows from (a) that $\text{range}(d_e)$ is uniformly recursive. It is a finite set because $d_e(s)$ changes only if a new element is enumerated in some set E_k, $k < e$, which happens only finitely often. Thus $\lim_{s\to\infty} d_e(s)$ exists and is finite. $\text{range}(d_e)$ and $\text{range}(d_{e'})$ are disjoint for $e \neq e'$ since $\langle -,-\rangle$ is injective.

(c) If $d_e(s)$ is enumerated into A at stage $s+1$, then e is declared "passive," so $d_e(s)$ is fixed at all later stages.

(d) This follows from (a).

(e) Clearly, A is r.e. Suppose for the sake of contradiction that A is recursive. Then there exists e with $\overline{A} = W_e$. By (a) and (b), there is a stage s such that $d_e(s') = d_e(s)$ for all $s' \geq s$. By construction, $d_e(s)$ is enumerated into A iff it is enumerated into W_e. This contradicts the hypothesis $\overline{A} = W_e$. □

CLAIM 2. *For all $e \geq 1$, we have the following:*

(a) $\psi_{\tau_{e,1}}$ and $\psi_{\tau_{e,2}}$ *are uniformly partial recursive.*

(b) *If e is always "active," then $\psi_{\tau_{e,1}}$ witnesses that* $\text{ic}_\psi(x : A) \leq e$ *for all* $x \in \text{range}(d_e)$.

(c) *If e is eventually "passive," then $\psi_{\tau_{e,2}}$ witnesses that* $\text{ic}_\psi(x : A) \leq e$ *for all* $x \in \text{range}(d_e)$.

Proof. (a) This follows from Claim 1(b).

(b) If e is always "active," then $\text{range}(d_e) \cap A = \emptyset$; thus $\psi_{\tau_{e,1}}$ is A-consistent and $\psi_{\tau_{e,1}}(x) = 0 = \chi_A(x)$ for all $x \in \text{range}(d_e)$.

(c) If e is declared "passive" at stage $s+1$, then $A \cap \text{range}(d_e) = \{d_e(s)\}$. Thus $\psi_{\tau_{e,2}}$ is A-consistent and $\psi_{\tau_{e,2}}(x) = \chi_A(x)$ for all $x \in \text{range}(d_e)$. □

Let ψ_e^s denote the finite portion of ψ_e defined at the end of stage s.

CLAIM 3. *For all $s = 2\langle k, t\rangle + 1$, we have the following:*

(a) *For all i, $1 \leq i \leq |M_k|$, $\psi_{p_{k,i}}^s$ is an A-consistent function.*

(b) *For all i, $1 \leq i \leq |M_k|$, if $\sigma_{k,s+1}(i) = 1$, then* $\operatorname{dom}(\psi_{p_{k,i}}^{s+1}) = \{x : l(x) \leq t\}$.

(c) *For all x, $l(x) < \operatorname{len}(k, s+1)$, if $x \notin R(k,s)$, then there exists i, $1 \leq i \leq |M_k|$, with $\sigma_{k,s+1}(i) = 1$ and $\psi_{p_{k,i}}^{s+1}(x) = \chi_{A_{s+1}}(x)$.*

Proof. (a) We use Claim 1(d) and the fact that $\psi_{p_{k,i}}$ is defined at stage $s+1$ only for arguments less than s. If $e < k$ and $\psi_{p_{k,i}}(d_e(s'))$ is defined, then $\psi_{p_{k,i}}(d_e(s')) = \perp$, so there is no problem with consistency. If $e \geq k$ and $\psi_{p_{k,i}}(d_e(s')) = \chi_{A_{s+1}}(d_e(s'))$ is defined at stage $s+1 > s'$, then either e is already "passive," so $\chi_{A_{s+1}}(d_e(s')) = \chi_A(d_e(s'))$, or e is "active" and we define $d_e(s+1)$ at stage $s+1$ such that $l(d_e(s+1)) > l(d_e(s'))$. In the latter case, we get $\psi_{p_{k,i}}(d_e(s')) = \chi_{A_{s+1}}(d_e(s')) = \chi_A(d_e(s')) = 0$.

(b) and (c) are shown by induction on s. Consider stage $s+1 = 2\langle k,t\rangle + 2$. If no new element is enumerated in E_k after exactly t steps, then $\sigma_{k,s+1} = \sigma_{k,s}$ and (b) and (c) follow from the induction hypothesis and the definition of $\psi_{p_{k,i}}$ at stage $s+1$.

Now assume that x enters E_k after exactly t steps. If case (a) occurs, the claim follows from the induction hypothesis. If case (b) occurs, we have $x \notin R(k,s)$ and $\operatorname{len}(k,s) \leq l(x) < t$. We have $\sigma_{k,s+1} = \operatorname{succ}(\sigma_{k,s})$, so $\sigma_{k,s+1}(i') = \sigma_{k,s}(i')$ for all $i' > i = \min\{j : \sigma_{k,s+1}(j) = 1\}$.

If $\sigma_{k,s+1}(i') = 0$ for all $i' > i$, then $s+1$ is the first stage s', where $\sigma_{k,s'}(i) = 1$. This means that $\psi_{p_{k,i}}^s = \lambda x.\uparrow$ and $\psi_{p_{k,i}}^{s+1}(z) = \chi_{A_s}(z) = \chi_{A_{s+1}}(z)$ for all z such that $l(z) \leq t$ and $z \notin R(k,s)$.

If there is $i' > i$ with $\sigma_{k,s+1}(i') = 1$, then there exists a greatest stage $s' < s$ with $\sigma_{k,s'}(i) = 1 \;\wedge \sigma_{k,s'+1}(i) = 0$. Then we have $\sigma_{k,s'+1}(i') = \sigma_{k,s+1}(i')$ for all $i' > i$ and we have $\sigma_{k,s'+1}(i') = \sigma_{k,s+1}(i') = 0$ for all $i' < i$. By induction hypothesis, we get for all x with $l(x) < \operatorname{len}(k, s'+1)$ that if $x \notin R(k,s')$, then there exists j with $\sigma_{k,s'+1}(j) = 1$ and $\psi_{p_{k,j}}^{s'+1}(x) = \chi_{A_{s'+1}}(x) = \chi_A(x)$. (The second equality holds by part (a).)

Since $R(k,s') \subseteq R(k,s)$, it only remains to consider x with $\operatorname{len}(k,s'+1) \leq l(x) < \operatorname{len}(k, s+1) = t+1$. Since $\sigma_{k,s'}(i) = 1$, it follows by induction hypothesis that $\operatorname{dom}(\psi_{p_{k,i}}^s) = \operatorname{dom}(\psi_{p_{k,i}}^{s'}) = \{x : l(x) < \operatorname{len}(k,s'+1)\}$. Thus $n = \min\{l(z) : z \notin \operatorname{dom}(\psi_{p_{k,i}}^s)\} = \operatorname{len}(k, s'+1)$, and at stage $s+1$, we define $\psi_{p_{k,i}}^{s+1}(z) = \chi_A(z)$ for all $z \notin R(k,s)$ such that $\operatorname{len}(k,s'+1) = n \leq l(z) < t+1 = \operatorname{len}(k,s+1)$. For $z \in R(k,s)$ and $l(z) \leq t$, we have $\psi_{p_{k,i}}^{s+1}(z) = \perp$. This completes the proof of (b) and (c). □

CLAIM 4. *For almost all x, $\operatorname{ic}_\psi(x : A) \leq \log C(x) + 2$.*

Proof. Let $k \geq 1$ be minimal such that $x \in E_k$. If $x \in R(k,s)$ for some s, then by Claim 2, we get $\operatorname{ic}_\psi(x : A) < k$. If $x \notin R(k,s)$ for all s, then let $\sigma_k = \lim_{s\to\infty} \sigma_{k,s}$. By Claim 3, there exists $i, 1 \leq i \leq |M_k|$, such that $\sigma_k(i) = 1 \;\wedge\; \psi_{p_{k,i}}(x) = \chi_A(x)$. Furthermore, $\psi_{p_{k,i}}$ is total recursive and A-consistent, so $\operatorname{ic}_\psi(x : A) \leq k$. Since $E_1 = \emptyset$, we have $k > 1$ and $x \notin E_{k-1}$, so $2^{k-1} - 2 \leq C(x)$, i.e., $k \leq \log(C(x)+2)+1 \leq \log C(x) + 2$ for all x with $C(x) \geq 2$. □

What happens for $\overline{\operatorname{ic}}$? Of course, the ICC also fails for $\overline{\operatorname{ic}}$. It even fails in a much stronger way because, in contrast to Theorem 4.1, $\overline{\operatorname{ic}}$ can be arbitrary small, as we now show.

THEOREM 4.3. *For every recursive function f, there is an r.e. nonrecursive set A such that*

$$f(\overline{\operatorname{ic}}(x : A)) \leq C(x) \quad \textit{for almost all } x.$$

Proof. We may assume that f is strictly increasing. As above, it suffices to define a partial recursive function $\psi(p,x)$ such that $f(\overline{\mathrm{ic}}_\psi(x:A)) \le C(x)$ for almost all x and A is nonrecursive. This leads to the following requirements for all $i \ge 1$:

(N_i)

$$(\forall x) \quad [C(x) < f(i+1) \;\Rightarrow\; (\exists p \in \{0,1\}^i) \quad [\chi_A \text{ extends } \psi_p \text{ and } \psi_p(x) = \chi_A(x)]];$$

$$(P_i) \qquad\qquad W_i \neq \overline{A}.$$

These can be satisfied by an easy finite-injury construction. Fix an enumeration of $E_i = \{x : C(x) < f(i+1)\}$ for all i.

During the construction, we have for each i a current $p_i \in \{0,1\}^i$ which satisfies (N_i) for all x that have been currently enumerated into E_i. If some x with $\psi_{p_i}(x) = 0$ is later enumerated into A, then ψ_{p_i} is no longer A-consistent and we have to choose a new p_i. Since we have 2^i candidates for p_i, we can afford $2^i - 1$ injuries.

Therefore, we are allowed to enumerate a diagonalization witness x into A at stage s for the sake of (P_i), only if x has not yet appeared in any E_j with $j \le i$. Clearly, (P_i) can still be satisfied. Furthermore, (N_i) is injured at most i times. Since $i \le 2^i - 1$ for all $i \ge 1$, every (N_i) will eventually be satisfied. □

Remark. In the course of the construction, at most $2^{f(i+1)} - 1$ elements are not allowed to be enumerated into A by (P_i). Hence we can fix in advance a set J_i of $2^{f(i+1)}$ witnesses for (P_i) and guarantee that one of them will be successful. Therefore, we can also modify the construction and satisfy the following requirements (P_i') instead of (P_i) for any fixed r.e. set B:

$$(P_i') \qquad\qquad i \in B \;\Leftrightarrow\; J_i \cap A \neq \emptyset.$$

Then we get $B \le_d A$. If we choose $B = K$, this shows that there is a d-complete set which satisfies the condition of the theorem. Since we need to enumerate at most one element of J_i into A, we get that $A \le_{wtt(1)} B$, i.e., A is wtt-reducible to B by a one-query reduction. Thus every r.e. wtt-degree contains a set A as in the theorem. It can be shown that this does not hold for r.e. tt-degrees.

5. R.e. sets with hard instances. While we have shown in the last section that the ICC fails for some r.e. nonrecursive sets, it is interesting to find out whether there are properties of r.e. sets which imply the existence of hard instances. We consider this question for classes of complete sets and simple sets. Indeed, in most cases, it turns out that such sets must have hard instances, which is a partial resurrection of the ICC.

Buhrman and Orponen [2], [10, Exercise 7.40] proved that the set of all random strings $R = \{x : C(x) \ge l(x)\}$ satisfies $\mathrm{ic}(x : R) \ge l(x) - O(1)$ for all $x \in R$. (Actually, their result also holds for $\overline{\mathrm{ic}}$ instead of ic.) Using the observation

$$(*) \qquad \text{If } A \le_m B \text{ via } f, \quad \text{then } \mathrm{ic}(x : A) \le \mathrm{ic}(f(x) : B) + O(1) \quad \text{for all } x$$

and the fact that R is co-r.e., they concluded that every m-complete set A has hard instances in its complement. They asked whether the hard instances can be chosen from A instead of $\overline{A}$. (This is, of course, impossible in the $\overline{\mathrm{ic}}$ version.) The next result gives a positive answer.

THEOREM 5.1. *There is an r.e. set A with* $\mathrm{ic}(x : A) \ge l(x)$ *for infinitely many* $x \in A$.

Proof. Uniformly in n, we enumerate $A \cap \{0,1\}^n$ as follows. Let $x_1, \ldots, x_{2^n}$ be a listing of all strings of length n in lexicographical order.

Construction:

Step 0: Enumerate x_1 into A; let $i = 1$, $I = \{0,1\}^{\leq n-1}$, $J = \{1, \dots, 2^n\} - \{1\}$.

Step $s+1$: If there is a program $p \in I$ such that

(a) $U_s(p, x_j) \in \{0,1\}$ for some $j \in J$ or

(b) $U_s(p, x_j) = \perp$ for all $j \in J$,

then choose the least such p, let $I = I - \{p\}$, and do the following:

In case (a), enumerate x_j into A iff $U_s(p, x_j) = 0$. Let $J = J - \{j\}$.

In case (b), let $i = \min(J)$. Enumerate x_i into A and let $J = J - \{i\}$.

End of Construction.

At the end of step 0, we have $|I| = |J| = 2^n - 1$. In all later steps, an element of I is removed iff an element of J is removed. Thus at the end of each step, we have $|I| = |J|$. Also, if case (b) occurs, then $\min(J)$ exists (since at that point, $|J| > 0$). Note that the value of $\chi_A(x_j)$ is fixed when j is removed from J.

Let i_0, I_0, and J_0 be the final values of i, I, and J in the above construction and choose s_0 such that $i = i_0$, $I = I_0$, and $J = J_0$ in all steps $t \geq s_0$. Suppose for the sake of contradiction that $\text{ic}(x_{i_0} : A) < n$ via $p \in \{0,1\}^{\leq n-1}$.

If $p \notin I_0$, then there is a stage $s \leq s_0$ when p was removed from I. If p was removed in case (a) via j, then $U(p, x_j) \neq \chi_A(x_j)$. If p was removed in case (b), then $U(p, x_{i_0}) = \perp$. Hence p does not witness that $\text{ic}(x_{i_0} : A) < n$, which is a contradiction.

If $p \in I_0$, then $|J_0| = |I_0| \geq 1$ and there is $t > s_0$ such that $U_t(p, x) \in \{0, 1, \perp\}$ for all $x \in J_0$. Hence at stage $t+1$, either case (a) or (b) occurs and $|I_0|$ decreases, contradicting the choice of s_0.

Thus we have $\text{ic}(x_{i_0} : A) \geq n = l(x_{i_0})$ and clearly $x_{i_0} \in A$. Since this holds for all n, the theorem is proved. □

Using (*), we get the following corollary.

COROLLARY 5.2. *For every m-complete set A, there is a constant c such that*

$$\text{ic}(x : A) \geq C(x) - c \quad \textit{for infinitely many } x \in A.$$

This result also holds for a much weaker reducibility, as we now show.

THEOREM 5.3. *For every wtt-complete set A, there is a constant c such that*

$$\text{ic}(x : A) \geq C(x) - c \quad \textit{for infinitely many } x \in A.$$

Proof. Suppose that A is a wtt-complete set. We enumerate an auxiliary r.e. set B and a uniformly r.e. sequence $\{E_n\}_{n \in \mathcal{N}}$ with $|E_n| \leq 2^n$. Then there is a partial recursive function $\psi : \{0,1\}^* \times \{0,1\}^* \to \mathcal{N}$ such that $\psi(\{0,1\}^n, \epsilon) = E_n$. Hence $C_\psi(x) \leq n$ for all $x \in E_n$ and there is a constant c, independent of n, such that $C(x) \leq n + c$ for all $x \in E_n$. Thus it suffices to satisfy the following requirement for all n:

$$(R_n) \qquad (\exists x \in E_n \cap A) \quad [\text{ic}(x : A) \geq n - 1].$$

By the recursion theorem and the fact that A is wtt-complete, we can assume that we are given in advance the index of a wtt-reduction from B to A, i.e., a Turing reduction Φ and a total recursive use bound g such that for all x, $\chi_B(x) = \Phi^A(x)$ and in the computation of $\Phi^A(x)$, every query is less than $g(x)$. (In more detail, there are recursive functions h and k such that for all e, W_e is wtt-reducible to A via $\Phi_{h(e)}$ with use bound $\varphi_{k(e)}$. For each e, an r.e. set $B = B_e$ is enumerated using $\Phi = \Phi_{h(e)}$ and $g = \varphi_{k(e)}$ in the construction below. By the s-m-n theorem, there is a recursive function f such that $B_e = W_{f(e)}$. By the recursion theorem, there is an index e_0 such that $W_{e_0} = W_{f(e_0)}$. Then $B = B_{e_0}$ is the required set.)

Each (R_n) is satisfied independently from the other requirements; therefore, for the following, fix n and let $x_1 = \langle n, 1\rangle, \ldots, x_{2^n} = \langle n, 2^n\rangle$, $m = \max\{g(x_i) : 1 \leq i \leq 2^n\}$, and $I = \{p : l(p) < n-1\}$. We enumerate E_n and $B \cap \{x_1, \ldots, x_{2^n}\}$ in steps $i = 0, \ldots, 2^n$ as follows.

Construction:

Step 0: Let $s_0 = 0$ and $E_n = \emptyset$.

Step $i+1$: Search for the least $s \geq s_i$ such that we have the following:

(1) $\Phi_s^{A_s}(x_j) = 1$ with use less than $g(x_j)$ for $j = 1, \ldots, i$ and $\Phi_s^{A_s}(x_j) = 0$ with use less than $g(x_j)$ for $j = i+1, \ldots, 2^n$.

(2) For each $x \in E_n$, there is $p \in I$ such that we have the following:

(2.1) $U_s(p, z)$ is defined for all $z \leq m$.

(2.2) $(\forall z \leq m)[U_s(p,z) \neq \perp \Rightarrow U_s(p,z) = \chi_{A_s}(z)]$.

(2.3) $U_s(p, x) = 1$.

Let $s_{i+1} = s$. Enumerate x_{i+1} into B and compute some $y \leq m$ with $y \in A - A_{s_{i+1}}$. (Note that y exists because otherwise $\Phi^A(x_{i+1}) = 0 \neq 1 = \chi_B(x_{i+1})$. We can find y by enumerating A.) Let CONS be the set of all $p \in I$ which satisfy conditions (2.1) and (2.2) for $s = s_{i+1}$. If $U_s(p, y) = \perp$ for all $p \in$ CONS, then enumerate y into E_n.

Goto step $i+2$.

End of Construction.

By construction, we have $|E_n| \leq 2^n$ and $E_n \subseteq A$. We want to argue that in some step of the construction, the search does not terminate. Since $\chi_B = \Phi^A$, this can only happen if condition (2) is not satisfied for any sufficiently large s. However, this means that $\mathrm{ic}(x : A) \geq n-1$ for some $x \in E_n$.

Consider the value of CONS $\subseteq I$ after each terminating step. We show that a new program enters CONS or a program is removed from CONS forever. Since there are at most $|I| < 2^{n-1}$ programs which may at some point become members of CONS, it follows that there are less than $2 \cdot 2^{n-1} = 2^n$ terminating steps, which completes the proof.

Note that if a program p is removed from CONS at some stage s, then there is y such that $U_s(p, y) = 0$ and $\chi_{A_s}(y) = 1$. Thus p cannot enter CONS again at any later stage.

Suppose that step $i+1$ terminates and consider the current value of CONS and of y at the end of this step. There are two cases:

(a) $U_s(p, y) = \perp$ for all $p \in$ CONS. Then y is enumerated into E_n, so in the next step, a new program must enter CONS such that condition (2.3) is satisfied for $x = y$.

(b) $U_s(p, y) \neq \perp$ for some $p \in$ CONS. Hence $U_s(p, y) = 0$ and since $\chi_{A_{s_{i+2}}}(y) = 1$, p is removed from CONS if the next step terminates. □

By a similar proof, one can show that every bounded-truth-table-complete (btt-complete) set has hard instances w.r.t. $\overline{\mathrm{ic}}$. We have noted in the remark following Theorem 4.3 that this is no longer true for d-complete sets, but we can show that it still holds for Q-complete sets.

Recall that A is *Q-complete* if it is r.e. and there is a recursive function g such that for all x,

$$x \in K \Leftrightarrow W_{g(x)} \subseteq A.$$

See [12, p. 281] for more information on Q-reducibility.

THEOREM 5.4. *Every Q-complete set A has hard instances, even w.r.t. $\overline{\text{ic}}$.*

Proof. Suppose that A is Q-complete. As in the previous proof, we enumerate an auxiliary r.e. set B and an r.e. sequence of finite sets $\{E_n\}_{n\in\mathcal{N}}$ such that $|E_n| \leq 2^n$. It suffices to get infinitely many n such that there is $y \in E_n$ with $\overline{\text{ic}}(y : A) \geq n-2$.

By the recursion theorem and the Q-completeness of A, we may assume that we are given in advance a recursive function g such that $B \leq_Q A$ via g, i.e., for all x, $x \in B \Leftrightarrow W_{g(x)} \subseteq A$.

The first idea is to run a version of the previous construction. We keep a number x out of B and find $y \in W_{g(x)}$ which has not yet been enumerated into A. Then we enumerate y into E_n and wait until some A-consistent program p with $l(p) < n-2$ shows up and $U(p,y) = 0$. Then we enumerate x into B, which forces y into A and diagonalizes p.

However, this approach does not work because it might happen that after we enumerate y into E_n, y is also enumerated into A, and *after that*, $U(p,y) = 1$ is defined. Then we cannot diagonalize p by enumerating x into B, but we have incremented $|E_n|$. Since this can happen an arbitrary finite number of times, we run into conflict with the requirement $|E_n| \leq 2^n$.

Therefore, we use the following modification. For each n, if $E_n \neq \emptyset$, then we enumerate y into E_n only if y has been previously enumerated into E_{n+1}, and then we proceed according to the first idea. If y is later enumerated into A, we get a diagonalization for $n+1$ instead of n, which is also fine.

Now we turn to the formal details. Let $I_n = \{p : l(p) < n-2\}$. $p \in \{0,1\}^*$ is called *A-consistent* at stage $s+1$ if, for all $z \leq s$, either $U_s(p,z)$ is undefined or $U_s(p,z) = \chi_{A_s}(z)$.

We maintain the following invariant for all n, s, and y:

If $E_n \neq \emptyset$ at stage $s+1$, then enumerate y into E_n only if $P(n,s,y)$ holds, where $P(n,s,y) \Leftrightarrow y \in E_{n+1} - A_s$, $E_n \subseteq A_s$, and there is $p \in I_{n+1}$ which is A-consistent at stage $s+1$ and $U_s(p,y) = 0$.

As a consequence of this invariant, it already follows that $|E_n| \leq 2^n$. Suppose that $E_n \neq \emptyset$ and we enumerate y into E_n at stage $s+1$. Then we enumerate the next element into E_n only after y has been enumerated into A, and hence the program $p \in I_{n+1}$ which had witnessed the condition $P(n,s,y)$ is diagonalized and can never again be A-consistent. Since $|I_{n+1}| < 2^{n-1}$, it follows that we will enumerate at most $1 + 2^{n-1}$ elements into E_n. In particular, $|E_n| \leq 2^n$ for all n.

We say that n is *saturated* at stage $s+1$ if for every $y \in E_n$, there is $p \in I_n$ such that p is A-consistent at stage $s+1$ and $U_s(p,y) = \chi_{A_s}(y)$. The goal of the construction is to produce infinitely many n such that each of them is only finitely often saturated. This implies at once that there are infinitely many $y \in E_n$ with $\overline{\text{ic}}(y : A) \geq n-2$, and we are done.

To achieve this goal, we construct a sequence $d_0 < d_1 < d_2 < \cdots$ and satisfy the following requirements

(R_i) The interval $[d_i, d_{i+1})$ contains an n which is only finitely often saturated.

The d_i's are constructed by recursive approximation. The value of d_i may change finitely often and eventually stabilizes. Some additional variables are needed for bookkeeping. For each i, there is a finite set T_i containing the set of all x's which may be enumerated into B for the sake of (R_i). For each n, we have three variables active(n), cand(n), and source(n). active(n) is a Boolean flag which indicates if there is some

$y \in E_n - A_s$ to be enumerated into E_{n-1}; in this case, cand$(n) = y$ and source$(n) = x$ such that $x \notin B_s$ and $y \in W_{g(x),s}$.

We say that i *requires attention* at stage $s+1$ if one of the following conditions holds at the beginning of stage $s+1$:

(1) d_{i+1} is undefined.

(2) d_{i+1} is defined and every $n \in [d_i, d_{i+1})$ is saturated at stage $s+1$.

Construction:

Stage 0: Let $d_0 = 0, d_{i+1} = \uparrow, T_i = \emptyset$ for all i. Let active$(n) = 0$ and $E_n = \emptyset$ for all n.

Stage $s+1$: For every n such that active$(n) = 1$ and cand$(n) \in A_s$ let active$(n) = 0$.

Let i be the least number which requires attention at stage $s+1$. If it requires attention through (1), then let $d_{i+1} = s+1$.

If it requires attention through (2), we then distinguish two cases:

(a) If there is a least $n \in (d_i, d_{i+1})$ such that active$(n) = 1$ and $E_{n-1} \subseteq A_s$, then enumerate cand(n) into E_{n-1} and let active$(n) = 0$. If $n - 1 = d_i$, then enumerate source(n) into B; otherwise, let active$(n-1) = 1$, cand$(n-1) =$ cand(n), and source$(n-1) =$ source(n).

(b) Otherwise, put $s+1$ into T_i and let $x = \min(T_i - B_s)$. Find the least s' such that $W_{g(x),s'} - A_s \neq \emptyset$ and let $y = \min(W_{g(x),s'} - A_s)$. Let active$(s+1) = 1$, cand$(s+1) = y$, source$(s+1) = x$, and enumerate y into E_{s+1}.

In both cases, let $T_i = T_i \cup \bigcup_{j>i} T_j$ and let $T_j = \emptyset$ and $d_j = \uparrow$ for all $j > i$.
End of Construction.

It easily follows by induction on s that our invariant is satisfied. Note that before we enumerate a new element into E_{n-1} via step (a), we require that $E_{n-1} \subseteq A_s$. If we enumerate an element via step (b), then the corresponding set was previously empty. Therefore, at each stage $s+1$, every E_n contains at most one element which is not in A_s. Now suppose that $E_{n-1} \neq \emptyset$ at the end of stage s and we enumerate an element y into E_{n-1} at stage $s+1$. Then case (a) occurred and $y =$ cand$(n) \notin A_s$ (since active$(n) = 1$). By the previous remarks, we have $E_{n-1} \subseteq A_s$. Since n is saturated at stage $s+1$, there is an A-consistent $p \in I_n$ such that $U_s(p, y) = \chi_{A_s}(y) = 0$. Thus $P(n-1, s, y)$ holds.

Hence it only remains to verify that requirement (R_i) is satisfied for all i. This is done by induction on i. By induction hypothesis, there is a least a stage s_0 such that $d_i = s_0$ is defined at stage s_0 and no $i' < i$ requires attention at any stage $s > s_0$. At the end of stage s_0, we have $E_{d_i} = \emptyset$ and $T_i = \emptyset$. We have shown above that the cardinality of E_{d_i} is always bounded by 2^{d_i}. Hence there exists $s_1 \geq s_0$ such that E_{d_i} does not change after stage s_1. Note that $E_{d_i} \subseteq A$ because each time that we enumerate y into E_{d_i}, we enumerate some x into B such that $x \in B \Leftrightarrow W_{g(x)} \subseteq A$ and $y \in W_{g(x)}$; thus we force y into A. Therefore, we can choose s_1 large enough such that $E_{d_i} \subseteq A_s$ for all $s \geq s_1$.

Suppose for the sake of contradiction that i requires attention infinitely often. We will argue that at some stage $s_2 > s_1$, a new element is enumerated into E_{d_i}, which contradicts the choice of s_1. There is a first stage $s+1 > s_1$ where i requires attention through (2). If case (a) applies, let $n_0 = n$; if case (b) applies, let $n_0 = s+1$. Let $x_0 =$ source(n_0) and $y_0 =$ cand(n_0). If $y \notin A$, then there is a stage $s' > s$ such that n_0 is the least $n > d_i$ with active$(n) = 1$ and $E_{n-1} \subseteq A_s$. In the following stages, when i requires attention, y will be enumerated into $E_{n_0}, E_{n_0-1}, E_{n_0-2}, \ldots$, and finally into E_{d_i}, which yields the desired contradiction. If $y \in A$, it might happen that y is enumerated into A before it arrives in E_{d_i}. Then, however, eventually a new candidate y' from $W_{g(x_0)}$ is chosen in case (2)(b), and a new attempt is started to bring y' into E_{d_i}. Again, it might happen that y' is enumerated into A before it arrives in E_{d_i}. However, this process cannot repeat infinitely often because otherwise $x_0 \notin B$ and hence there is some $y \in W_{g(x_0)} - A$. This y would in some iteration be chosen as a candidate which cannot be enumerated into A. Therefore, at some stage $s_2 + 1 > s_1$, some y is enumerated into E_{d_i}. Since $y \notin A_{s_2}$ and $E_{d_i} \subseteq A_{s_2}$, this implies that E_{d_i} increases, which is a contradiction.

Thus i requires attention only finitely often and (R_i) is satisfied. This completes the proof of the inductive step. □

Recall that A is *strongly effectively simple* if it is a coinfinite r.e. set and there is a total recursive function f such that for all e,

$$W_e \subseteq \overline{A} \Rightarrow \max(W_e) < f(e).$$

Since every strongly effectively simple set is Q-complete [12, Exercise III.6.21a], we get the following corollary.

COROLLARY 5.5. *Every strongly effectively simple set has hard instances, even w.r.t.* $\overline{\mathrm{ic}}$.

It is known that hyperhypersimple sets are not Q-complete [12, Thm. III.4.10], but we can still show that they have hard instances.

THEOREM 5.6. *Every hyperhypersimple set has hard instances, even w.r.t.* $\overline{\mathrm{ic}}$.

Proof. The basic idea of this proof is similar to that of Theorem 5.4. Assume that A is hyperhypersimple. We enumerate an r.e. sequence of finite sets $\{E_n\}_{n\in\mathcal{N}}$ such that $|E_n| \leq 2^n$. It suffices to get infinitely many n such that there is $y \in E_n$ with $\overline{\mathrm{ic}}(y : A) \geq n - 2$.

Let $I_n = \{p : l(p) < n - 2\}$. We initialize $E_n = \{n\}$ and may later enumerate elements from E_n into E_{n-1}. This time, we ensure that at any stage s, *at most two* elements of E_n belong to $\overline{A}_s$. We never enumerate an element twice into the same set. Furthermore, we enumerate x into E_n at stage $s + 1$ only if there is a program $p \in I_{n+1}$ which is A-consistent at stage $s + 1$ and $U_s(p, x) = 0$.

From this invariant, it already follows that $|E_n| \leq 2^n$. It is easy to see by induction on k that we enumerate the $(2k + 1)$st element into E_n at stage $s + 1$ only if there are at least k programs p from I_{n+1} which were A-consistent at some previous stage and are now diagonalized (i.e., for each such p, there is $z \in E_n \cap A_s$ such that $U_s(p, z) = 0$). Since there are less than 2^{n-1} programs in I_{n+1}, it follows that $|E_n| < 2 \cdot 2^{n-1} + 1 = 2^n + 1$.

As in the previous proof, we say that n is *saturated* at stage $s + 1$ if for every $y \in E_n$, there is $p \in I_n$ such that p is A-consistent at stage $s+1$ and $U_s(p, y) = \chi_{A_s}(y)$. We want to produce infinitely many n's which are only finitely often saturated.

To this end we construct for each e a sequence $d_0^e < d_1^e < \cdots$ such that for each i, $|\overline{A} \cap E_{d_i^e}| \geq 1$ or there is $n \in [d_i^e, d_{i+1}^e)$ which is only finitely often saturated. Suppose we have constructed at the end of stage s an initial segment of this sequence, say $d_0^e < \cdots < d_{m+1}^e$. Let $\mathrm{count}(n, s) = |\overline{A}_s \cap E_{n,s}|$. We extend this initial segment at stage $s + 1$ only if $\mathrm{count}(d_i^e, s) \geq 1$ for all $i \leq m$. In the end, we shall be able to argue that if the sequence is infinite, then there is a weak array which witnesses that A is not hyperhypersimple. Thus the sequence must be finite, say $d_0^e < \cdots < d_{m(e)+1}^e$, and there is $n \in [d_0^e, d_{m(e)+1}^e)$ which is only finitely often saturated. Also, since the strategy to extend the eth sequence is active at only finitely many stages, we can build an $(e+1)$st sequence with $d_0^{e+1} > d_{m(e)+1}^e$, which will also be finite and gives us another number that is only finitely often saturated, etc.

We assign priorities as follows. The definition of the eth sequence has higher priority than the definition of the e'th sequence if $e < e'$. The definition of the ith member of the eth sequence has higher priority than the definition of the i'th member if $i < i'$. In other words, we take the lexicographical ordering $<_{\text{lex}}$ on $\mathcal{N} \times \mathcal{N}$ as our priority ordering.

For technical reasons, we enumerate for each e a set M_e. When we are working on the eth sequence, we try to establish for each d_i^e an element $x \in E_{d_i^e} - A$. In M_e, we enumerate the current candidate for x.

We say that (e, i) *requires attention* at stage $s+1$ if one of the following conditions holds at the beginning of stage $s+1$.

(1) d_i^e is undefined and for all $j \in [0, i-1)$, $\text{count}(d_j^e, s) \geq 1$ and every $n \in [d_j^e, d_{j+1}^e)$ is saturated at stage $s+1$.

(2) d_i^e and d_{i+1}^e are both defined, $\text{count}(d_i^e, s) = 0$, and every $n \in [d_i^e, d_{i+1}^e)$ is saturated at stage $s+1$.

Construction:

Stage 0: Let $d_i^e =\uparrow$ and $M_e = \emptyset$ for all e and i, and let $E_n = \{n\}$ for all n.

Stage $s+1$: Choose the lexicographically least (e, i) which requires attention at stage $s+1$.

If it requires attention through (1), then let $d_i^e = s+1$ and let $d_j^{e'} =\uparrow$ for all $(e', j) >_{\text{lex}} (e, i)$. If $i = 0$ or $\text{count}(d_{i-1}^e, s) \geq 1$, then enumerate $s+1$ into M_e.

If it requires attention through (2), there is a least $n \in (d_i^e, d_{i+1}^e)$ such that $\text{count}(n-1, s) \leq 1$, and there is a least $x \in E_{n,s} - (A_s \cup M_{e,s} \cup E_{n-1,s})$, then enumerate x into E_{n-1}. If, in addition, $n - 1 = d_i^e$, then enumerate x into M_e. In any case, let $d_j^{e'} =\uparrow$ for all $(e', j) >_{\text{lex}} (e, i)$.

End of Construction.

It easily follows by induction on s that $\text{count}(n, s) \leq 2$ for all n and s, in particular, $|E_n \cap \overline{A}| \leq 2$. Also, we enumerate at stage $s+1$ an element x from E_n into E_{n-1} only if it does not yet belong to $E_{n-1} \cap A$ and n is saturated. In particular, there is a program $p \in I_n$ which is A-consistent at stage $s+1$ and $U_s(p, x) = 0$.

CLAIM. *For every e, there are only finitely many stages where (e, i) requires attention for some i.*

Proof. Suppose for the sake of contradiction that there exists a least e and infinitely many s's such that (e, i) requires attention at stage $s+1$ for some i. Then we argue that A is not hyperhypersimple. First, there is a least stage $s_0 \geq 1$ such that no (e', i') with $e' < e$ requires attention at any stage $s \geq s_0$. Then we define $d_0^e = s_0$ at stage s_0 and enumerate s_0 into M_e. By the choice of s_0, the value of d_0^e has stabilized. Note that all elements which have been previously enumerated into M_e are less than s_0 and so they do not matter for the following. By induction on $s \geq s_0$, it follows that $E_{n,s}$ contains at most one element from $M_{e,s} - A_s$ for all $n \geq d_0^e$.

We now distinguish two cases:

(a) If there is a least i such that (e, i) requires attention infinitely often, then there is a stage $s_1 \geq s_0$ where all d_j^e's with $j \leq i$ have stabilized. Thus (e, i) infinitely often requires attention through (2) and d_{i+1}^e tends to infinity. Then, however, it follows similarly to the previous proof that unboundedly many elements are eventually enumerated into $E_{d_i^e}$, which contradicts the fact that the cardinality of $E_{d_i^e}$ is bounded.

If (e, i) requires attention through (2) at any stage $s \geq s_1$, then $\text{count}(d_i^e, s) = 0$; thus an (e, j) with $j > i$ cannot require attention through (2) at any later stage $s' > s$

until a new element is enumerated into $E_{d_i^e}$ and $\mathrm{count}(d_i^e, s') = 1$. During that time, M_e does not change. (This is because, if (e, j) requires attention at stage $\tilde{s} + 1$, then $j \in \{i, i + 1\}$. If $(e, i + 1)$ requires attention, then it requires attention, through (1) and $\tilde{s} + 1$ is not enumerated into M_e as $\mathrm{count}(d_i^e, \tilde{s}) = 0$.) This guarantees that eventually a new element is enumerated into $E_{d_i^e}$ since there exist elements $z \in (\bigcup_{n>d_i^e} E_{n,s}) - (A \cup M_e \cup E_{d_i^e,s})$. Since $|E_{n,s} \cap (M_{e,s} - A_s)| \leq 1$ for $n \geq d_i^e$, it causes no problems to maintain the constraint that an element x is enumerated from E_n into E_{n-1} at stage $s + 1$ only if $x \notin (M_{e,s} \cup A_s)$.

(b) If for every i there are only finitely many stages (but at least one stage) where (e, i) requires attention, then it follows that the values d_i^e stabilize and form an infinite increasing sequence. Let d_i^e denote the final value. Since the sequence is infinite, it follows that $\lim_s \mathrm{count}(d_i^e, s) \geq 1$; thus $|E_{d_i^e} \cap \overline{A}| \geq 1$. From the actual construction, we get $|E_{d_i^e} \cap \overline{A}| = 1$ and $E_{d_i^e} \cap \overline{A} \subseteq M_e$.

Uniformly in i, we enumerate an r.e. set U_i as follows. For every stage $s + 1 \geq s_0$ where (e, i) is chosen, we enumerate $s + 1$ into U_i if case (1) applies and $s + 1$ is enumerated into M_e. If case (2) applies and x is the element enumerated into $E_{d_i^e}$, then we enumerate x into U_i.

Since each element enumerated into U_i is at the same time enumerated into M_e and is therefore blocked for the other sets, it follows that the U_i's are pairwise disjoint. By the remarks above, each U_i intersects $\overline{A}$. Thus A is not hyperhypersimple. This contradiction completes the proof of the claim. □

Thus for each e, there exists a maximal $m(e) \geq 0$ such that the value of $d^e_{m(e)+1}$ stabilizes and no (e, j) with $j > m(e) + 1$ requires attention at any sufficiently large stage. This means that there exists $n \in [d_0^e, d^e_{m(e)+1})$ which is only finitely often saturated. Thus there is $y \in E_n$ with $\overline{\mathrm{ic}}(y : A) \geq n - 2$. Clearly, we get infinitely many pairwise-different such y's. This completes the proof. □

The previous result does not hold for hypersimple sets since one can construct a hypersimple set that does not have hard instances. This can be done, e.g., by a direct modification of the proof of the next theorem.

Recall that A is *effectively simple* if it is a coinfinite r.e. set and there is a recursive function f such that for all e,

$$W_e \subseteq \overline{A} \Rightarrow |W_e| \leq f(e).$$

It is known that every effectively simple set is T-complete [12, Prop. III.2.18].

THEOREM 5.7. *There is an effectively simple set which does not have hard instances. In particular, there is a T-complete set which does not have hard instances.*

Proof. The construction in the proof of Theorem 4.2 is not combinable with the requirement of making A effectively simple. Therefore, we use a modified version, where we do not attempt to have the instance complexity as low as possible.

In the following, we outline the construction. A will be effectively simple for some f to be determined later. As in the proof of Theorem 4.2, we are given a uniformly r.e. sequence $\{E_k\}_{k\in\mathcal{N}}$ and we build a partial recursive function ψ such that for almost all k and each $x \in E_k$, there is some $p \in \{0, 1\}^k$ that witnesses that $\mathrm{ic}_\psi(x : A) \leq k$.

How do we define ψ_p? We will keep a list $S = S_k$ of programs of length k. The length of S will be fixed (depending on k). Furthermore, we have a pool $P = P_k$ of unused programs of length k. At the beginning, $|S| + |P| = 2^k$. During the construction, some of the programs in S may become inconsistent with A, in which case they are removed from S and new programs from P are inserted into S. There may also exist a "backup program" chosen from P.

The programs in S will be defined at x with a 0/1-value only if x was enumerated into E_k. The definition proceeds in circular order. The first program in S takes care of the first element which is enumerated into E_k, the second program takes care of the second element, and so on. In this way, we handle the first $|S|$ elements. Ideally, we would like to again have that the first program takes care of the $(|S|+1)$st element, etc. However, this does not work because as soon as a program is brought into play, we have to define it for larger and larger inputs. Thus it might happen that all of our programs are already defined (with output $\perp$) at x when x is enumerated into E_k as the $(|S|+1)$st element at stage s.

Thus we are using a program q from P which is still undefined everywhere and define it as $\chi_{A_s}(z)$ for all $z < s$; in particular, this covers all elements currently in E_k. For all larger values, we output $\perp$. q is called the current backup. We also suspend defining the programs in S until new elements $x \geq s$ are enumerated into E_k. Then we continue as above for the next $|S|$ such elements. After that, a new program from P is defined as the current backup in a similar way as q, and so on.

What is the advantage of that scheme? It is more robust against injuries which may happen when an element x with $\psi_p(x) = 0$ is later enumerated into A. In that case, only one $p \in S$ is destroyed. Also, only the $x \in E_k$ are critical because for $x \notin E_k$ we have $\psi_p(x) = \perp$. If p is destroyed, then we assign a new program from P as a substitute.

A crucial part in this process is the definition of the new backup q when a round has been completed at the beginning of stage s. Before we define ψ_q, we enumerate into A all $x < s$ which do not belong to any E_n with $n < g(k)$. This defines the current A_s. Here g is some fast-growing function to be determined later. Then we define $\psi_q(x) = \chi_{A_s}(x)$ for all $x < s$ and $\psi_q(x) = \perp$ otherwise.

We use the following strategy to make A effectively simple. If at the end of some stage s, we have $W_{e,s} \subseteq \overline{A}_s$ and $|W_{e,s}| > f(e)$, then choose an $x \in W_{e,s}$ which does not belong to any E_n with $n \leq g(e)$ and enumerate it into A. Note that x exists if we choose f large enough such that $f(e) \geq |E_0| + |E_1| + \cdots + |E_{g(e)}|$.

This completes the description of the construction. It remains to choose the parameters such that it works. We first count how many of the ψ_p's with $l(p) = k$ are used. Then we choose $|S_k|$, $|E_k|$, and g in such a way that the number of programs used is less than 2^k.

Let $m = \max\{i : g(i) \leq k\}$. Then for each $i \leq m$, there can be $\lceil |E_i|/|S_i| \rceil$ many rounds, and after each round, all programs in S_k may be destroyed (and have to be replaced by new ones from P_k). At this time, it is important that after the action of i, we immediately define the new programs that replace the former ones which have been destroyed. We can do this without any further enumeration of elements into A. There is no cascading effect which could blow up the number of injuries. Thus at most $|S_k| \sum_{i=1}^{m} \lceil |E_i|/|S_i| \rceil$ many programs of length k are ever injured while in S_k.

How many of the backup functions are destroyed? Note that this may happen each time when some $i < k$ acts, i.e., whenever i completes a round. Thus at most $\sum_{i=0}^{k-1} \lceil |E_i|/|S_i| \rceil$ many backup programs are destroyed.

The number of injuries resulting from making A effectively simple can be bounded by $m+k+1$. If we act for the sake of $W_{e,s} \cap A \neq \emptyset$ (which happens at most once), then a program from S_k can be destroyed only if $e \leq m$, and a current backup program can be destroyed only if $e < k$. To see the latter, note that if the current q is defined at $x \notin E_0 \cup \cdots \cup E_{g(k)}$, then $\psi_q(x) \in \{1, \perp\}$ because of the additional enumeration of elements into A which was performed when q was brought into play.

TABLE 1

r	m	btt	c	d	p	tt	wtt	Q	T
ic	×	×	×	×	×	×	×	×	
$\overline{\text{ic}}$	×	×	×					×	

Thus we need to ensure that for almost all k,

$$(+) \qquad 2^k > |S_k| \sum_{i=1}^{m} \lceil |E_i|/|S_i| \rceil + \sum_{i=1}^{k-1} \lceil |E_i|/|S_i| \rceil + m + k + 1.$$

Let $|S_k| = \lfloor 2^k/k \rfloor$, $g(k) = 2^k$, and $E_k = \{x : C(x) < 3k/2\}$, so $|E_k| < 2^{3k/2}$. Define the recursive function f by $f(e) = \lceil \sum_{i=0}^{g(e)} 2^{3i/2} \rceil$. The right-hand side of (+) is bounded above by

$$(2^k/k)(\log k)^2 \sqrt{k} + k^2 2^{k/2} + \log k + k + O(1),$$

which is less than 2^k for all sufficiently large k. With this choice of parameters, we get for almost all x, $C(x) \geq (3/2)(\text{ic}_\psi(x : A) - 1)$, i.e., $\text{ic}_\psi(x : A) \leq (2/3)C(x) + 1$. Thus A does not have hard instances. □

The previous results characterize the reducibilities $\leq_r$ with r $\in$ {m, btt, c, d, p, tt, wtt, Q, T} (cf. the figure in [12, p. 341] for the implications between these reducibilities) such that every r-complete set has hard instances, for both ic and $\overline{\text{ic}}$. In Table 1, we have marked the possible combinations.

Remark. The T-degrees of r.e. sets with hard instances do not coincide with any of the known degree classes. It can be shown that they form a proper subclass of the r.e. nonrecursive degrees and that they properly extend the array nonrecursive degrees. There is also an r.e. nonrecursive degree such that all of its r.e. sets have hard instances.

6. Open questions.

We have provided a comprehensive picture of instance complexity of r.e. sets. However, there are still interesting open questions left for further research:

(i) How low can the instance complexity of a T-complete set be? Is there a T-complete set A with $\text{ic}(x : A) \leq \log C(x) + O(1)$ for all x?

(ii) Does every T-degree that contains an r.e. set with hard instances also contain an r.e. set with hard instances w.r.t. $\overline{\text{ic}}$?

(iii) Study the instance complexity of non-r.e. sets, e.g., is there a set A such that $\text{ic}(x : A) \geq C(x) - O(1)$ for all x?

Acknowledgments. I would like to thank Lance Fortnow, William I. Gasarch, Carl Jockusch, Paul Vitányi, and the two referees for comments and corrections.

REFERENCES

[1] J. 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] H. BUHRMAN AND P. ORPONEN, *Random strings make hard instances*, in Proc. 9th Annual Conference on Structure in Complexity Theory, IEEE Computer Society Press, Los Alamitos, CA, 1994, pp. 217–222.

[3] C. CALUDE, *Information and Randomness*, Springer-Verlag, Berlin, 1994.

[4] G. J. Chaitin, *Information-theoretic characterizations of recursive infinite strings*, Theoret. Comput. Sci., 2 (1976), pp. 45–48.

[5] R. Downey, C. G. Jockusch, and M. Stob, *Array nonrecursive sets and multiple permitting arguments*, in Proc. Recursion Theory Week, K. Ambos-Spies, G. H. Müller, and G. E. Sacks, eds., Lecture Notes in Math. 1432, Springer-Verlag, Berlin, 1990, pp. 141–174.

[6] R. Downey, C. G. Jockusch, and M. Stob, *Array nonrecursive sets and genericity*, in Directions in Computability Theory, Cambridge University Press, Cambridge, UK, to appear.

[7] L. Fortnow and M. Kummer, *On resource-bounded instance complexity*, Theoret. Comput. Sci., to appear.

[8] K. Ko, *A note on the instance complexity of pseudorandom sets*, in Proc. 7th Annual Conference on Structure in Complexity Theory, IEEE Computer Society Press, Los Alamitos, CA, 1992, pp. 327–337.

[9] K. Ko, P. Orponen, U. Schöning, and O. Watanabe, *What is a hard instance of a computational problem?*, in Structure in Complexity Theory, A. Selman, ed., Lecture Notes in Comput. Sci. 223, Springer-Verlag, Berlin, 1986, pp. 197–217.

[10] M. Li and P. Vitányi, *An Introduction to Kolmogorov Complexity and Its Applications*, Springer-Verlag, Berlin, 1993.

[11] D. W. Loveland, *A variant of the Kolmogorov concept of complexity*, Inform. and Control, 15 (1969), pp. 510–526.

[12] P. Odifreddi, *Classical Recursion Theory*, North–Holland, Amsterdam, 1989.

[13] P. Orponen, *On the instance complexity of NP-hard problems*, in Proc. 5th Annual Conference on Structure in Complexity Theory, IEEE Computer Society Press, Los Alamitos, CA, 1990, pp. 20–27.

[14] P. Orponen, K. Ko, U. Schöning, and O. Watanabe, *Instance complexity*, J. Assoc. Comput. Mach., 41 (1994), pp. 96–121.

[15] R. I. Soare, *Recursively Enumerable Sets and Degrees*, Springer-Verlag, Berlin, 1987.

[16] J. Tromp, *On a conjecture by Orponen* + 3, in Descriptive Complexity, R. V. Book, E. Pednault, and D. Wotschke, eds., Dagstuhl-Seminar-Report, vol. 63, IBFI GmbH, Wadern, Germany, 1993, p. 20.

SIAM J. COMPUT.
Vol. 25, No. 6, pp. 1144–1170, December 1996

AN $o(n^3)$-TIME MAXIMUM-FLOW ALGORITHM*

JOSEPH CHERIYAN†, TORBEN HAGERUP‡, AND KURT MEHLHORN‡

Abstract. We show that a maximum flow in a network with n vertices can be computed deterministically in $O(n^3/\log n)$ time on a uniform-cost RAM. For dense graphs, this improves the previous best bound of $O(n^3)$.

The bottleneck in our algorithm is a combinatorial problem on (unweighted) graphs. The number of operations executed on flow variables is $O(n^{8/3}(\log n)^{4/3})$, in contrast with $\Omega(nm)$ flow operations for all previous algorithms, where m denotes the number of edges in the network. A randomized version of our algorithm executes $O(n^{3/2}m^{1/2}\log n + n^2(\log n)^2/\log(2+n(\log n)^2/m))$ flow operations with high probability.

For the special case in which all capacities are integers bounded by U, we show that a maximum flow can be computed deterministically using $O(n^{3/2}m^{1/2} + n^2(\log U)^{1/2} + \log U)$ flow operations and $O(\min\{nm, n^3/\log n\} + n^2(\log U)^{1/2} + \log U)$ time. We finally argue that several of our results yield parallel algorithms with optimal speedup.

Key words. network flow, maximum flow, graph algorithm, scaling, preflow-push algorithm, current-edge problem, dynamic tree

AMS subject classifications. 68P05, 68Q20, 68Q22, 68Q25, 68R05, 90B10, 90C35

1. Introduction. The fastest algorithm predating this paper for computing a maximum flow in a network with n vertices and m edges, even allowing randomization, has an expected running time of $O(\min\{nm\log n, nm + n^2(\log n)^2\})$ [CH95]. Despite intensive research for over three decades, no algorithm with a running time of $o(nm)$ has ever been reported for any combination of n and m. This is true even for networks with integer capacities, provided that the maximum capacity U is moderately large, say $U = \Omega(n)$ [AOT89].

Our main result is a maximum-flow algorithm that runs in $O(n^3/\log n)$ time. For dense networks with $m = \omega(n^2/\log n)$ this is $o(nm)$. We also slightly improve the best previous results for sparse networks with $m = o(n(\log n)^2)$ and $m = \omega(n\log n/\log\log n)$ and match the best previous results for other ranges of m through simpler algorithms. Our algorithms are *strongly polynomial*; this means, roughly speaking, that the running time is bounded by a polynomial in the number of vertices in the network independently of the capacities of the edges (see [GLS88] for a more careful definition). Table 1 below summarizes the running times of our algorithms for different combinations of n and m.

Subsequently to our work, King, Rao, and Tarjan [KRT94] extended the range of network densities for which the performance of randomized maximum-flow algorithms can be matched by deterministic algorithms. Their deterministic algorithm runs in $O(nm\log_{m/(n\log n)} n)$ time on networks with n vertices and $m > n\log n$ edges. This running time is $O(nm)$ when $m \geq n^{1+\epsilon}$ for some fixed $\epsilon > 0$, while it is strictly larger than our running time for all other network densities.

*Received by the editors November 20, 1991; accepted for publication (in revised form) February 8, 1995. This research was partially supported by EC ESPRIT II Basic Research Actions Program contract 3075 (project ALCOM). A preliminary version of this paper was presented at the 17th International Colloquium on Automata, Languages and Programming (ICALP) in July 1990.

†Department of Combinatorics and Optimization, University of Waterloo, Waterloo, ON N2L 3G1, Canada.

‡Max-Planck-Institut für Informatik, D-66123 Saarbrücken, Germany (torben@mpi-sb.mpg.de).

TABLE 1

Strongly polynomial maximum-flow algorithms for different combinations of n, the number of vertices, and m, the number of edges. The first column gives the order of the bound on the running time of each algorithm, the second column indicates the range of network densities for which the algorithm is superior to the other algorithms, the third column states whether the algorithm is deterministic or randomized, and the last column gives the source of the algorithm and a corresponding theorem in the present paper. The new results appear in lines 2 and 5 of the body of the table.

Running Time	Range	Model	Source
$nm \log n$	$m \le n\frac{\log n}{\log\log n}$	deterministic	[ST83] and this paper, Theorem 8.1(a)
$\frac{n^2(\log n)^2}{\log(2+n(\log n)^2/m)}$	$n\frac{\log n}{\log\log n} \le m \le n(\log n)^2$	randomized	this paper, Theorem 8.1(c)
nm	$n(\log n)^2 \le m \le n^{5/3}\log n$	randomized	[CH95] and this paper, Theorem 8.1(c)
nm	$n^{5/3}\log n \le m \le \frac{n^2}{\log n}$	deterministic	[Al90] and this paper, Theorem 8.1(b)
$\frac{n^3}{\log n}$	$\frac{n^2}{\log n} \le m \le n^2$	deterministic	this paper, Theorem 8.2

Our algorithm is based on earlier work in [CH89], [GT88], and [AO89], all of which in turn use the generic maximum-flow algorithm of Goldberg and Tarjan [GT88], which works by manipulating a so-called *preflow* [Ka74] in the given network. We design an extension of the generic algorithm, called the *incremental generic algorithm*, which uses a new operation called *add edge*. The new algorithm manipulates a preflow in a subnetwork and, as the execution progresses, gradually adds the remaining edges to the current subnetwork.

Adding the edges in the order of decreasing capacities allows instances of the incremental generic algorithm to save on the number of operations on flow variables. In particular, the number of flow operations executed by our main algorithm is $O(n^{8/3}(\log n)^{4/3})$. To the best of our knowledge, all previous algorithms execute $\Omega(nm)$ flow operations. Using randomization, we can do even better: a maximum flow can be computed using $O(n^{3/2}m^{1/2}\log n + n^2(\log n)^2/\log(2+n(\log n)^2/m))$ flow operations with high probability. In fact, our deterministic algorithm is obtained from the randomized algorithm by applying a derandomization technique due to Alon [Al90]. Our analysis of flow operations is based on a novel potential argument.

The bottleneck in our algorithms turns out to be a simple combinatorial problem on a dynamically changing (unweighted) graph, that of repeatedly identifying the so-called *current edge* of a given vertex. Indeed, given a sufficiently efficient solution to the current-edge problem, the running time of each of our algorithms would match the number of flow operations. A straightforward solution to the current-edge problem contributes $\Theta(nm)$ time to the running time of the maximum-flow algorithms. The idea behind our improvement of this bound for dense networks, by a factor of $\Theta(\log n)$, is to represent the graph by its adjacency matrix and to partition the matrix into $1 \times \lfloor \log n \rfloor$ submatrices. A submatrix can be processed in constant time by table look-up during the search for a current edge.

Our ideas also apply to networks with integer capacities. For networks with integer capacities bounded by U, one of the fastest algorithms known is the wave scaling algorithm of [AOT89], which runs in time $O(nm + n^2(\log U)^{1/2} + \log U)$.

This algorithm refines the excess scaling algorithm of [AO89], whose running time is $O(nm + n^2 \log U)$. For neither algorithm is a better bound than the running time known for the number of flow operations. We give incremental versions of both algorithms and show how to replace the term nm by $\min\{nm, n^3/\log n\}$ in the bound for the running time and by $n^{3/2}m^{1/2}$ in the bound for the number of flow operations.

The paper is organized as follows. Basic definitions are given in §2. The incremental generic algorithm and the current-edge problem are introduced in §3. The incremental excess scaling and wave scaling algorithms for networks with integer capacities are described in §§4 and 5, respectively. Section 6 discusses solutions to the current-edge problem. The strongly polynomial algorithm is presented in §7 and analyzed in §§7 and 8. Section 9 discusses parallel versions of our algorithms, and §10 states a number of open problems. Readers with an exclusive interest in the strongly polynomial algorithm can skip §§4 and 5 almost entirely. The only material in these sections needed later is the definition of γ-fooling height and its properties (F1)–(F5), given after the proof of Lemma 4.1.

2. Definitions and notation.

For every set V and every $e = (v, w) \in V \times V$, let *tail*$(e) = v$, *head*$(e) = w$, and *rev*$(e) = (w, v)$. v and w are the *tail* of e and the *head* of e, respectively. A *network* is a tuple $G = (V, E, cap, s, t)$, where (V, E, cap) is an edge-weighted directed graph, *cap* maps each edge in E to a nonnegative real number called its *capacity*, and s and t are distinct vertices in V called the *source* and the *sink*, respectively. We assume that E is *symmetric* (i.e., *rev*$(e) \in E$ for all $e \in E$) and without self loops (i.e., $v \neq w$ for all $(v, w) \in E$). In order to make the notation less cumbersome, we omit one pair of brackets from expressions such as "*cap*$((v, w))$."

A *preflow* in G is a function $f : E \to \mathbb{R}$ with the following properties:

(1) $f(rev(e)) = -f(e)$ for all $e \in E$ (antisymmetry constraint),

(2) $f(e) \leq cap(e)$ for all $e \in E$ (capacity constraint),

(3) $\sum_{e \in E: head(e)=v} f(e) \geq 0$ for all $v \in V\backslash\{s\}$ (nonnegativity constraint).

A preflow f in G is a *flow* if $\sum_{e \in E: head(e)=v} f(e) = 0$ for all $v \in V\backslash\{s, t\}$ (flow conservation constraint). The *value* of f is $\sum_{e \in E: head(e)=t} f(e)$, i.e., the net flow into t, and a *maximum flow* in G is a flow in G of maximum value. An edge $e \in E$ is *residual* (with respect to f) if $f(e) < cap(e)$. A *push* on e of *value* $c \in \mathbb{R}$ is an increase in $f(e)$ by c. The push is *saturating* iff $f(e) = cap(e)$ afterwards. A push on an edge (v, w) is also called a push *out of* v and a push *into* w. A *labeling* of G is a function $d : V \to \mathbb{N} \cup \{0\}$. The labeling is *valid* for G and a preflow f in G exactly if $d(v) \leq d(w) + 1$ for every edge $(v, w) \in E$ that is residual with respect to f. Our algorithms operate with the concept of an *undirected edge*, i.e., a pair $\{v, w\}$, where $(v, w) \in E$, which intuitively we identify with the pair $\{(v, w), (w, v)\}$ of two antiparallel directed edges. For all symmetric subsets E' of E, let $\overline{E'} = \{\{v, w\} : (v, w) \in E'\}$ be the corresponding set of undirected edges. A push on an undirected edge $\{v, w\} \in \overline{E}$ is a push on one of the edges (v, w) or (w, v). The *capacity* of an undirected edge $\{v, w\}$ is defined as $cap(\{v, w\}) = cap(v, w) + cap(w, v)$. We assume without loss of generality that $cap(\{v, w\}) > 0$ for all $\{v, w\} \in \overline{E}$.

Our algorithms are formulated in the traditional model for the study of problems on networks. They use two data types for numerical values: *integer* and *flow value*. Capacities and flow values are represented by objects of type *flow value*, on which the only allowed arithmetical operations are addition and subtraction, and all other quantities are represented by objects of type *integer*, on which we allow addition, subtraction, multiplication, and integer division. In addition, we assume for both data types standard operations for comparison, data movement, the constant 1, etc. For n-vertex input networks, we allow integers of absolute value $n^{O(1)}$; i.e., we allow a word size of $O(\log n)$ bits. We charge constant time for each basic operation on either type (uniform cost measure). In keeping with common usage, we employ the term "flow operation" to mean any operation on objects of type *flow value.* In our randomized algorithms we assume in addition that generating a random integer takes constant time. More precisely, we assume that for every given integer k with $1 \leq k \leq n$, a random integer drawn from the uniform distribution over $\{1, \ldots, k\}$ and independent of all other such random integers can be obtained in constant time.

We use "log" to denote the logarithm to base 2, and we assume lists to be implemented as a data type with operations *first* and *pop* (among others). Given a list L, *first*(L) returns the first element of L, and *pop*(L) removes the first element of L and returns it.

3. The incremental generic algorithm.

In this section, we generalize the generic maximum-flow algorithm of [GT88] by extending it to include one additional operation, *add edge.*

The goal of the algorithm is to compute a maximum flow in a network $G = (V, E, cap, s, t)$. Let $n = |V|$ and $m = |E|$. In order to avoid trivialities, we assume that $m \geq n \geq 3$. Let $V^+ = V \backslash \{s, t\}$. The main variables used by the incremental generic algorithm are the following:

(1) A network $G^* = (V, E^*, cap^*, s, t)$, where $E^* \subseteq E$ is symmetric and cap^* is the restriction of cap to E^*. G^* is the *current network*, on which the algorithm operates. $E^* = \emptyset$ initially, and the edges in E are gradually added to E^*.

(2) A preflow $f : E^* \rightarrow \mathbb{R}$, which gradually evolves into a maximum flow in G.

(3) A labeling $d : V \rightarrow \mathbb{N} \cup \{0\}$, valid for f and G^*.

An edge $(v, w) \in E^*$ is called *eligible* exactly if it is residual with respect to f and $d(v) = d(w) + 1$. For all $e \in E^*$, the *residual capacity* of e is defined as $rescap(e) = cap(e) - f(e)$, and for all $v \in V$, the *excess* of v is defined as $excess(v) = \sum_{e \in E^* : head(e) = v} f(e)$.

We now briefly review the generic maximum-flow algorithm of [GT88], which works on the complete network throughout the execution. The labeling d and the preflow f are initialized as follows: $d(s) = n$, $d(v) = 0$ for all $v \in V \backslash \{s\}$, $f(e) = cap(e)$ for all edges e with tail s, $f(e) = -cap(rev(e))$ for all edges e with head s, and $f(e) = 0$ for all other edges e. Note that the initial labeling is valid for the initial preflow. The algorithm repeatedly picks some vertex $v \in V^+$ with positive excess and performs either a push out of v or a relabeling of v. More precisely, if there is an eligible edge with tail v then a push is performed on one such edge, and if there is no eligible edge with tail v then $d(v)$ is increased by one. This maintains the validity of d, which is crucial for the analysis; cf. Lemmas 3.3 and 3.6 below. The algorithm terminates when there is no vertex in V^+ with positive excess.

In the incremental generic algorithm, we start with $d(s) = n$, $d(v) = 0$ for all $v \in V \backslash \{s\}$, and $E^* = \emptyset$, and we gradually add the edges in E to E^*. This creates a problem, however. The validity of d is endangered whenever an edge (v, w) with $d(v) > d(w) + 1$ is added to E^*. We overcome this difficulty by adopting a more conservative rule than that of [GT88] for sending flow out of a vertex. Namely, define the *visible excess*, *excess*$^*(v)$, of a vertex $v \in V$ by

$$excess^*(v) = excess(v) - \sum_{e \in E \backslash E^* : tail(e) = v} cap(e)$$

and relabel a vertex or send flow out of it only if its visible excess is positive and remains nonnegative (this use of visible excess is in some ways similar to the use of "available excess" in [AOT89]). We will show below (Lemma 3.1) that this rule guarantees that $excess^*(v) \geq 0$ whenever $d(v) > 0$. In particular, when an edge (v, w) with $d(v) > d(w)$ is added to E^*, it can be saturated immediately using the excess available at v, so that d remains valid. We now give the details.

The algorithm maintains the current network G^*, the preflow f, the labeling d, and the functions $excess(v)$ and $excess^*(v)$. Although the functions *excess* and *excess** in principle can be computed from f and E^*, efficiency dictates that they must be represented explicitly. In the description of the algorithm, however, we omit this trivial elaboration.

Since f is by definition antisymmetric, low-level flow manipulation is carried out by the procedure

```
PROCEDURE setflow(e: edge; c: real);
  $f(e) := c$; $f(rev(e)) := -c$;
```

with the special case

```
PROCEDURE saturate(e: edge);
  setflow(e, cap(e));
```

The main routines of the incremental generic algorithm and the algorithm itself follow.

```
PROCEDURE push(e: edge; c: real);
Precondition: $e = (v,w) \in E^*$, $v \in V^+$, e is eligible, and $0 < c \leq \min\{excess^*(v), rescap(e)\}$.
  setflow(e, $f(e) + c$);

PROCEDURE relabel(v: vertex);
Precondition: $v \in V^+$, $excess^*(v) > 0$, and no edge in $E^*$ with tail v is eligible.
  $d(v) := d(v) + 1$;

PROCEDURE add edge({v,w}: undirected edge);
Precondition: $\{(v,w),(w,v)\} \subseteq E \backslash E^*$.
  $E^* := E^* \cup \{(v,w),(w,v)\}$;
  if $d(v) > d(w)$ then saturate(v,w) fi;
  if $d(w) > d(v)$ then saturate(w,v) fi;

PROCEDURE generic initialize;
  for all $e \in E$ do setflow(e,0) od; (* zero flow is default for new edges *)
  for all $v \in V \backslash \{s\}$ do $d(v) := 0$ od; $d(s) := n$;
  $E^* := \emptyset$;

INCREMENTAL GENERIC ALGORITHM:
  generic initialize;
  while $\max\{excess^*(v) : v \in V^+\} > 0$ or $E^* \neq E$
  do
    Execute some push, relabel, or add edge operation whose precondition is satisfied;
    (* there always is one *)
  od.
```

An execution of $relabel(v)$ is called a *relabeling* of v. Define a push to be *regular* if it does not take place during a call of *add edge*; there are at most m nonregular pushes.

Note the special status of the source and the sink: no regular pushes are performed out of s or t, nor are they ever relabeled.

We next show the partial correctness of the algorithm (i.e., if it terminates, it will do so with the correct result) and give a few additional properties. Our proof is similar to the correctness proof of the generic algorithm of [GT88]. In stating invariants for the algorithm, we consider *push*, *relabel*, and *add edge* to be atomic operations; i.e., we ignore possible violations of the invariants while these routines are being executed. We also implicitly restrict attention to the part of the execution that follows the initialization.

LEMMA 3.1. *At all times during an execution of the incremental generic algorithm and for all* $v \in V^+$ *if* $d(v) > 0$ *then* $excess^*(v) \geq 0$.

Proof. We use induction on the number of steps executed by the algorithm. The claim is clearly true immediately after the initialization and after a relabeling of v (by the precondition of the *relabel* operation). A push into v does not decrease the visible excess of v, and a regular push out of v does not decrease it below zero (by the precondition of the *push* operation). Finally, observe that the execution of an *add edge* operation cannot decrease the visible excess of any vertex. □

LEMMA 3.2. *At all times during the execution,*

(a) f *is a preflow,*

(b) d *is a valid labeling.*

Proof. (a) and (b) hold initially, and they are not invalidated by calls of *push* or *relabel* (cf. [GT88, Lem. 3.1]). Furthermore calls of *add edge* are easily seen to preserve (b). The only remaining issue is that for some $v \in V\backslash\{s\}$, a saturating push on an edge (v, w) performed during a call of *add edge* might invalidate the nonnegativity constraint $excess(v) \geq 0$. However, when the push takes place, $d(v) > d(w) \geq 0$, and it follows from Lemma 3.1 that $excess\ (v) \geq 0$ after the push. □

LEMMA 3.3. *Suppose that the algorithm terminates. Then, at termination,* f *is a maximum flow in* G.

Proof. At termination, $G^* = G$ and $excess(v) = excess^*(v) = 0$ for all $v \in V^+$. Thus f is a flow in G. If f is not maximum, then, by a classical theorem of Ford and Fulkerson [FF62, Cor. 5.2], there exists an augmenting path with respect to f, i.e., a simple path in G from s to t all of whose edges are residual with respect to f. Since $d(s) = n$, $d(t) = 0$, and the length of a simple path in G is at most $n - 1$, this contradicts the validity of d. □

LEMMA 3.4. *An ineligible edge* $(v, w) \in E^*$ *can become eligible only during a relabeling of* v.

Proof. An edge $e = (v, w)$ is ineligible exactly if either $rescap(e) = 0$ or $d(v) \leq d(w)$. The residual capacity of e can increase from zero to a positive value only during a push on $rev(e)$. But $d(w) > d(v)$ at the time of such a push on $rev(e)$; i.e., e is ineligible after the push. Thus only a relabeling of v can make e eligible. □

Lemmas 3.5 and 3.6 below are analogous to Lemmas 3.5 and 3.7 of [GT88], respectively. We include them for the sake of completeness.

LEMMA 3.5. *For all* $v \in V^+$ *and at all times during the execution, if* $excess(v) > 0$, *then there is a simple path in* G^* *from* v *to* s *all of whose edges are residual with respect to* f.

Proof. Let S be the set of vertices reachable from v in G^* by a path all of whose edges are residual with respect to f. We need to show that $s \in S$. Assume otherwise. The choice of S implies that $f(u, w) \leq 0$ for all edges $(u, w) \in E^*$ with $u \notin S$ and $w \in S$. Using the antisymmetry of f, we obtain $\sum_{w \in S} excess(w) =$

$\sum_{(u,w)\in E^*:u\in V,w\in S} f(u,w) = \sum_{(u,w)\in E^*:u\in V\setminus S,w\in S} f(u,w) + \sum_{(u,w)\in E^*:u\in S,w\in S} f(u,w) \le 0$. Since $excess(w) \ge 0$ for all $w \in V\setminus\{s\}$, we conclude that $excess(v) = 0$, a contradiction. □

LEMMA 3.6. *For all $v \in V$ and at all times during the execution, $d(v) \le 2n - 1$. In particular, the total number of relabelings executed by the algorithm is $< 2n^2$.*

Proof. Let $v \in V^+$ with $excess(v) > 0$ be arbitrary. By Lemma 3.5 there is a simple path from v to s in G^* all of whose edges are residual with respect to f. Since $d(s) = n$, d is valid, and a simple path consists of at most $n - 1$ edges, this implies that $d(v) \le 2n-1$. Thus no vertex v with $d(v) = 2n-1$ can ever be relabeled. □

We discuss instances of the incremental generic algorithm in §§4, 5, and 7. For all instances an efficient implementation of the following *current-edge abstract data type* is important: the task is to maintain two functions, $r : E \to \{0,1\}$ and $h : V \to \{0,\dots,2n-1\}$, under the operations specified below. An edge $(v,w) \in E$ is called *admissible* if $r(v,w) = 1$ and $h(v) = h(w) + 1$. For $v \in V$, let $E(v) = \{(v,w) \in E : (v,w) \text{ is admissible}\}$.

Init;
Sets $h(v) := 0$ for all $v \in V\setminus\{s\}$, $h(s) := n$, and $r(v,w) := 0$ for all $(v,w) \in E$;

Spush(v,w);
Precondition: $v \in V$ and $(v,w) \in E(v)$.
Sets $r(v,w) := 0$ and $r(w,v) := 1$;

Npush(v,w);
Precondition: $v \in V$ and $(v,w) \in E(v)$.
Sets $r(v,w) := r(w,v) := 1$;

Relabel(v);
Precondition: $v \in V$, $E(v) = \emptyset$, and $h(v) < 2n - 1$.
Executes $h(v) := h(v) + 1$;

Add edge$(\{v,w\})$;
Precondition: $\{v,w\} \in \overline{E}$ and $r(v,w) = r(w,v) = 0$.
Sets $r(v,w) := 1$ if either $h(v) < h(w)$ or $(h(v) = h(w)$ and $cap(v,w) > 0)$;
Sets $r(w,v) := 1$ if either $h(w) < h(v)$ or $(h(v) = h(w)$ and $cap(w,v) > 0)$;

ce(v);
Precondition: $v \in V$.
Returns some $(v,w) \in E(v)$ if $E(v) \ne \emptyset$, *nil* otherwise;

For $q \in \mathbb{N}$, denote by $T_{ce}(n,m,q)$ the time needed to execute any legal sequence of one *Init* operation followed by q *Spush*, *Npush*, *Relabel*, *Add edge*, and *ce* operations. Note that such a sequence contains at most m *Add edge* operations, since at all times after a call *Add edge*$(\{v,w\})$ we have $r(v,w) = 1$ or $r(w,v) = 1$. Implementations of the current-edge data type are discussed in §§6 and 8. The algorithms of §§4 and 5 use the current-edge data type as follows: *generic initialize* calls *Init*; a saturating and a nonsaturating regular push on an edge e call *Spush*(e) and *Npush*(e), respectively; *relabel*(v) calls *Relabel*(v); and *add edge*$(\{v,w\})$ calls *Add edge*$(\{v,w\})$. For the sake of simplicity we do not explicitly mention these calls in the description of the algorithms, and we consider them to form atomic entities with the calling operations.

With this interface, it is easy to verify that the following holds throughout the execution: $h(v) = d(v)$ for all $v \in V$, and $r(v,w) = 1$ if and only if (v,w) is residual, for all $(v,w) \in E^*$. To see that the latter condition holds after a call *add edge*$(\{v,w\})$, recall that the call saturates (v,w) if $d(v) > d(w)$, saturates (w,v) if $d(w) > d(v)$, and leaves the flow on (v,w) at zero if $d(v) = d(w)$. Thus an edge $(v,w) \in E^*$ is eligible if and only if it is admissible, and for all $v \in V$ a call $ce(v)$ returns an eligible edge with tail v if there is one and *nil* otherwise. The function *ce* is used by the flow algorithms to find eligible edges on which to push flow and to test whether a vertex can be relabeled.

4. The incremental excess scaling algorithm.

In this section, we describe an incremental excess scaling algorithm for the case in which all edge capacities are integers bounded by $U \geq 1$. The algorithm is an adaptation of the excess scaling algorithm of Ahuja and Orlin [AO89] to the incremental paradigm.

The execution of the algorithm is divided into *phases* parameterized by the value of a *scaling parameter* Δ (of type *flow value*). The algorithm repeatedly chooses a vertex $v \in V^+$ with $excess^*(v) \geq \Delta$ and minimal $d(v)$ and either pushes flow on an edge (v,w) or relabels v. When there are no more vertices $v \in V^+$ with $excess^*(v) \geq \Delta$, the current phase ends, Δ is replaced by $\Delta/2$, all edges $(v,w) \in E \backslash E^*$ with $cap(\{v,w\}) \geq \Delta/\beta$ are added to E^*, and the next phase begins. Here β is a positive integer, which we will later fix at $\lfloor (m/n)^{1/2} \rfloor \geq 1$. The complete program follows.

```
INCREMENTAL EXCESS SCALING ALGORITHM:
  generic initialize;
  $L :=$ list of all undirected edges in $\overline{E}$ ordered by decreasing capacities;
  $\Delta := 2^{\lfloor \log U \rfloor}$;
  while $\Delta \geq 1$
  do
    while $L \neq \emptyset$ and $cap(first(L)) \geq \Delta/\beta$ do add edge(pop(L)) od;
    while $\max\{excess^*(v) : v \in V^+\} \geq \Delta$
    do
      Among the vertices $v \in V^+$ with $excess^*(v) \geq \Delta$, choose $v$ as one with minimal
      $d(v)$;
      if $ce(v) = nil$
      then relabel(v)
      else $e := ce(v)$; push(e, $\min\{\Delta, rescap(e)\}$) fi;
    od;
    $\Delta := \Delta/2$;
  od.
```

If and when the algorithm terminates, we have $E^* = E$ (since $\beta \geq 1$) and *excess*$^*(v) < 1$ for all $v \in V^+$. Since all flow values computed by the algorithm are integral, this implies that $excess^*(v) = 0$ for all $v \in V^+$ at that point. Thus the algorithm is an instance of the incremental generic algorithm and hence is partially correct. The algorithm refines the excess scaling algorithm of Ahuja and Orlin [AO89]; in fact, for $\beta = \infty$, i.e., if all edges are added before the first phase, the two algorithms are identical.

Denote by ♯*pushes*, ♯*relabels*, ♯*add edge*, and ♯*ce* the total number of regular pushes, relabelings, calls to *add edge*, and calls to *ce*, respectively, executed by the algorithm. We first analyze ♯*pushes* using a potential argument inspired by that of [CH89, Lem. 2]. Although part of this argument appears identically in [CH95], we repeat it in full for the reader's convenience. The argument is used to bound the number of regular saturating as well as the number of nonsaturating pushes. The analysis of the excess scaling algorithm by Ahuja and Orlin [AO89] also uses a potential argument. Their argument, however, applies only to nonsaturating pushes.

For $v \in V$ and $i = 1, 2, \ldots$, denote by $\deg_i(v)$ the number of edges with head v added to E^* between phase $i-1$ and phase i (for $i = 1$: before the first phase). Further, for $i = 1, 2, \ldots$, let $m_i = \sum_{v \in V} \deg_i(v)$.

Fact 4.1. At the time of a regular push on an edge $e = (v, w)$, e is eligible, the value of the push is $\le \Delta$, and if $w \in V^+$ then *excess*$^*(w) < \Delta$ immediately before the push.

LEMMA 4.1. *For all $v \in V^+$, excess$^*(v) < 2\Delta$ at the beginning of phase* 1, *and excess$^*(v) < 2\Delta + 2\deg_i(v)\Delta/\beta$ at the beginning of phase i for $i \ge 2$.*

Proof. Consider first the case $i = 1$. A call *add edge*$(\{s, v\})$ increases *excess*$^*(v)$ by at most $U < 2\Delta$, and there is at most one such call for each vertex $v \in V^+$. All other calls of *add edge* before phase 1 leave the flow unchanged since all vertices v except s have $d(v) = 0$. This completes the case $i = 1$. For $i \ge 2$ observe that *excess*$^*(v) < 2\Delta$ for all $v \in V^+$ before the calls of *add edge* between phases $i-1$ and i and that each call *add edge*$(\{u, v\})$ between these phases adds at most $2\Delta/\beta$ to *excess*$^*(v)$. Thus *excess*$^*(v) < 2\Delta + 2\deg_i(v)\Delta/\beta$ at the beginning of phase i for all $i \ge 2$. □

For $\gamma \ge 1$, call a regular push on an edge (u, v) a *γ-push* if $|\{w \in V : d(w) = d(v)\}| \ge \gamma$ at the time of the push, and define the *γ-fooling height* $d_\gamma(v)$ of a vertex $v \in V$ as follows: if $V = \{v_1, \ldots, v_n\}$, then

$$d_\gamma(v) = \max_{i_1 \ge d(v_1), \ldots, i_n \ge d(v_n)} |\{k \in \mathbb{Z} : 0 \le k < d(v) \text{ and } |\{j : i_j = k\}| \ge \gamma\}|.$$

Intuitively, $d_\gamma(v)$ counts the maximum number of "dense virtual distance levels" between v and t, where a vertex v_j 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.

d_γ has the following properties, named for future reference:

(F1) $\forall v \in V : 0 \le d_\gamma(v) \le n/\gamma$;

(F2) $\forall v \in V : d(v) = 0 \Rightarrow d_\gamma(v) = 0$;

(F3) $\forall u, v \in V : d(u) > d(v) \Rightarrow d_\gamma(u) \ge d_\gamma(v)$;

(F4) $\forall u, v \in V : (d(u) > d(v) \text{ and } |\{w \in V : d(w) = d(v)\}| \ge \gamma) \Rightarrow d_\gamma(u) > d_\gamma(v)$;

(F5) A relabeling of a vertex $v \in V^+$ increases $d_\gamma(v)$ by at most 1 and does not increase $d_\gamma(w)$ for any $w \in V \backslash \{v\}$.

Define the *normalized value* of a push as the value of the push divided by Δ.

LEMMA 4.2.

(a) *For all $\gamma \ge 1$ the total normalized value of all γ-pushes is at most* $(2n^2 \log U + 2nm/\beta)/\gamma + 4n^2$;

(b) *there are $O(nm/\beta + n^2(\log U + 1))$ nonsaturating pushes*;

(c) *there are $O(nm/\beta + n^2\beta + n^2 \log U)$ saturating pushes.*

Proof.

(a) Define the potential function

$$\Phi = \sum_{v \in V^+ : excess^*(v) \le 2\Delta} \frac{excess^*(v)}{\Delta} \cdot d_\gamma(v) + \sum_{v \in V^+ : excess^*(v) > 2\Delta} \frac{excess^*(v)}{\Delta} \cdot \frac{n}{\gamma}.$$

At the start of phase 1, $\Phi = 0$ (by Lemma 4.1, property (F2), and the fact that $d(v) = 0$ for all $v \in V^+$ at the start of phase 1), and $\Phi \ge 0$ always (by Lemma 3.1 and property (F2)). Φ does not increase due to regular pushes (by Fact 4.1 and properties (F1) and (F3)), and a relabeling increases Φ by at most 2 (by property (F5)). For $i \geq 2$, the change of Δ and the addition of edges between phases $i-1$ and i increase Φ by at most $(2n + 2m_i/\beta) \cdot n/\gamma$ (by Lemma 4.1 and property (F1)). Consequently, and by Lemma 3.6, the total increase, and hence also the total decrease, in Φ is at most $(2n^2 \log U + 2nm/\beta)/\gamma + 4n^2$. Finally, note that each γ-push of normalized value c causes Φ to decrease by at least c (by Fact 4.1 and property (F4)).

(b) Every push is a 1-push and every nonsaturating push has normalized value 1. The bound now follows from part (a), applied with $\gamma = 1$.

(c) Call a regular push on an edge (u, v) *small* if its value is less than Δ/β, call it *terminal* if $|\{w \in V : d(w) = d(v)\}| < \beta$ at the time of the push, and partition the regular saturating pushes into three classes: (1) small pushes, (2) nonsmall terminal pushes, and (3) nonsmall nonterminal pushes. We bound the number of pushes in each class separately.

(1) We have $cap(\{v, w\}) \geq \Delta/\beta$ for each $\{v, w\} \in \overline{E^*}$. Hence between any two small saturating pushes on a fixed undirected edge there is a nonsaturating push on that edge. Therefore the number of small saturating pushes is at most m plus the number of nonsaturating pushes, and the bound follows from part (b).

(2) By Lemma 3.4, each terminal push out of a vertex $v \in V$ is followed by fewer than β saturating pushes out of v before the next relabeling of v. Summing over all $v \in V$ and all possible values of $d(v)$, this gives at most $2n^2\beta$ terminal saturating pushes.

(3) The normalized value of a nonsmall push is at least $1/\beta$, and each nonterminal push is a β-push. An application of part (a) with $\gamma = \beta$ now shows that there are at most $2n^2 \log U + 2nm/\beta + 4n^2\beta$ regular nonsmall nonterminal pushes. □

We sum up the findings in the following theorem.

THEOREM 4.1. *A maximum flow in a network with n vertices, m edges, and integer capacities bounded by $U \geq 1$ can be computed deterministically using $O(q)$ flow operations and $O(q) + T_{ce}(n, m, q)$ time, where $q = O(n^{3/2}m^{1/2} + n^2 \log U)$.*

Proof. Put $\beta = \lfloor (m/n)^{1/2} \rfloor$ and note that β can be computed within the stated resources. Sorting the undirected edges by their capacities takes $O(m \log m) = O(n^{3/2}m^{1/2})$ time, the execution of *generic initialize* is no more expensive, and the initial value of Δ can be computed in $O(m + \log U)$ time. There are $\lfloor \log U \rfloor + 1$ phases, in each of which a number of undirected edges is added to $\overline{E^*}$. This takes $O(\log \beta) = O(\log n)$ time per undirected edge (for the multiplication of its capacity by β) and hence $O(m \log n)$ time altogether. Using simple data structures described in [AO89], the selection of v in the second inner while loop of the algorithm can be implemented to run in constant time per vertex selection plus $O(n)$ time per phase. Over the whole algorithm, this adds up to $O(\sharp pushes + \sharp relabels + n(\log U + 1))$ time. Since $\sharp pushes = O(n^{3/2}m^{1/2} + n^2 \log U)$ (by Lemma 4.2), $\sharp relabels = O(n^2)$ (by Lemma 3.6), $\sharp ce = O(\sharp pushes + \sharp relabels)$, and $\sharp add\ edge \leq m$, both the total number of operations executed on the current-edge data structure and the total time spent outside this data structure are $O(n^{3/2}m^{1/2} + n^2 \log U)$. The claim follows. □

In the next section, we show how the wave scaling technique of [AOT89] can be combined with the incremental approach to reduce the value of q in Theorem 4.1 to $O(n^{3/2}m^{1/2} + n^2(\log U)^{1/2} + \log U)$.

5. The incremental wave scaling algorithm.

The incremental wave scaling algorithm is an adaptation of the wave scaling algorithm of [AOT89] to the incremental paradigm. As in the previous section all edge capacities are integers bounded by

$U \geq 1$, and the execution is divided into phases parameterized by Δ, with edge additions taking place between phases.

The incremental wave scaling algorithm makes use of the procedures *stack push relabel* and *wave.* A call *stack push relabel*(v) pushes flow out of v until either the visible excess of v is zero or there are no eligible edges with tail v, in which case v is relabeled. Also, when *stack push relabel*(v) considers an eligible edge (v, w) and w has visible excess Δ or more, *stack push relabel* is first called recursively with argument w in order to "clear the way" for the push on (v, w). The procedure *wave* orders the vertices in V^+ by decreasing values of the functions d and then steps through the ordered list of vertices, calling *stack push relabel* for each vertex in turn. An important property of processing the vertices in this order is that once a call *stack push relabel*(v) in *wave* has terminated, the visible excess of v remains unchanged until the end of the call of *wave.* The algorithm employs the two procedures as follows: in each phase, it first uses *stack push relabel* to reduce the individual visible excess of each vertex in V^+ below Δ and then *wave* to reduce Σ^* below $n\Delta/l$, where $l > 0$ is a parameter to be chosen later and $\Sigma^* = \sum_{v \in V^+} \max\{excess^*(v), 0\}$. Although not strictly accurate, it is helpful to think of Σ^* as the total visible excess.

```
PROCEDURE stack push relabel(v: vertex);
  while $excess^*(v) > 0$ and $ce(v) \neq nil$
  do
    $w := head(ce(v))$;
    if $w \neq t$ and $excess^*(w) \geq \Delta$
    then stack push relabel(w)
    else (* $w = t$ or $excess^*(w) < \Delta$ *)
      $push((v, w), \min\{excess^*(v), \Delta, rescap(v, w)\})$;
    fi;
  od;
  if $excess^*(v) > 0$ then relabel(v) fi;

PROCEDURE wave;
  $J :=$ list of all vertices $v \in V^+$ ordered by decreasing values of $d(v)$;
  while $J \neq \emptyset$
  do stack push relabel(pop(J)) od;

INCREMENTAL WAVE SCALING ALGORITHM:
  generic initialize;
  $L :=$ list of the undirected edges in $\overline{E}$ ordered by decreasing capacities;
  $\Delta := 2^{\lfloor \log U \rfloor}$;
  while $\Delta \geq 1$
  do
    while $L \neq \emptyset$ and $cap(first(L)) \geq \Delta/\beta$ do add edge(pop(L)) od;
    (* A *)
    while $\max\{excess^*(v) : v \in V^+\} \geq \Delta$
    do
      Choose $v \in V^+$ with $excess^*(v) \geq \Delta$;
      stack push relabel(v);
    od;
    (* B *)
    while $\Sigma^* \geq n\Delta/l$ do wave od;
    (* C *)
    $\Delta := \Delta/2$;
od.
```

The parameters l and β will be chosen later. The incremental wave scaling algorithm differs in two respects from the wave scaling algorithm of [AOT89]: (1) it is incremental; and (2) instead of beginning each phase with a sequence of *waves*, i.e., calls of *wave*, we first reduce the visible excess of every vertex below Δ before executing the waves. This is necessary because the addition of edges at the beginning of a phase may cause individual excesses to be very large. In return, it is not necessary to reduce individual excesses after the waves as in [AOT89]—this is taken care of by the next phase.

We next elucidate the relationship between the incremental excess scaling algorithm of §4 and the incremental wave scaling algorithm of this section. Without the "wave loop," i.e., the loop between labels B and C, the two algorithms are basically the same. The wave loop reduces Σ^* below $n\Delta/l$. This allows us to replace the $n^2 \log U$ term in Lemma 4.2 by $n^2 \log U/l$ and thus yields an improved bound on the number of nonsaturating pushes. On the other hand, the wave loop brings about additional cost proportional to $n^2 l$ (since each wave has cost $\Theta(n)$ that cannot be accounted for by the techniques of the previous section, and since the number of waves is essentially proportional to nl; cf. Lemma 5.2). Choosing $l = (\log U)^{1/2}$ balances the two contributions and reduces the $n^2 \log U$ term in Theorem 4.1 to $n^2(\log U)^{1/2}$ in Theorem 5.1.

We now analyze the incremental wave scaling algorithm. If and when the algorithm terminates, we have $E^* = E$ (since $\beta \geq 1$) and $excess^*(v) < 1$ for all $v \in V^+$. Thus the algorithm is an instance of the incremental generic algorithm and therefore partially correct. Also, Fact 4.1 holds for it. Lemma 4.1 can be sharpened to the following.

Fact 5.1. For $i \geq 1$, the following bounds on total and individual visible excesses hold during phase i:

(a) at label A, $excess^*(v) < 2\Delta$ for $i = 1$ and for all $v \in V^+$, and $\Sigma^* < 2n\Delta/l + 2m_i\Delta/\beta$ for $i > 1$;

(b) at label B and until the end of the phase, $excess^*(v) < 2\Delta$ for all $v \in V^+$.

Denote by $\sharp$*stack push relabels* and by $\sharp$*waves* the number of calls of *stack push relabel* and of *wave*, respectively, and by $\sharp$*pushes* the number of regular pushes executed by the algorithm. We first show the following refinement of Lemma 4.2.

LEMMA 5.1.

(a) *For all $\gamma \geq 1$, the total normalized value of all γ-pushes is at most* $2n^2 \log U/(l\gamma) + 2nm/(\beta\gamma) + 4n^2$;

(b) $\sharp$*stack push relabels* $= O(nm/\beta + n^2 + n^2 \log U/l + n \cdot \sharp$*waves*$)$;

(c) $\sharp$*pushes* $= O(nm/\beta + n^2\beta + n^2 \log U/l + n \cdot \sharp$*waves*$)$.

Proof.

(a) We use the same potential function Φ as in the proof of Lemma 4.2(a). At the start of phase 1, $\Phi = 0$ (by Fact 5.1(a), property (F2) of γ-fooling height, and the fact that $d(v) = 0$ for all $v \in V^+$ at the beginning of phase 1). By the same argument as in the proof of Lemma 4.2(a), $\Phi \geq 0$ always, Φ does not increase due to regular pushes, and a relabeling increases Φ by at most 2. For $i \geq 2$, the change of Δ and the addition of edges between phases $i - 1$ and i increase Φ by at most $(2n/l + 2m_i/\beta) \cdot n/\gamma$ (by Fact 5.1(a) and property (F1)). Consequently, the total decrease in Φ is at most $2(n/l) \cdot (n/\gamma) \cdot \log U + 2(m/\beta) \cdot (n/\gamma) + 4n^2$. Finally, note that each γ-push of normalized value c causes Φ to decrease by at least c (by property (F4)).

(b) Define a call *stack push relabel*(v) to be *potent* if *excess*$^*(v) \geq \Delta$ at the time of the call. Since each nonpotent call of *stack push relabel* is made directly by *wave*, there can be at most $n \cdot \sharp$*waves* such calls. Also at most $2n^2$ calls *stack push relabel*(v) end with a relabeling of v (by Lemma 3.6). A potent call *stack push relabel*(v) that does not end with a relabeling of v, finally, carries out pushes out of v of total normalized value at least 1. Since every push is a 1-push, an application of part (a) with $\gamma = 1$ now shows the number of such calls to be $O(nm/\beta + n^2 + n^2 \log U/l)$.

(c) There is at most one nonsaturating push of value $< \Delta$ per call of *stack push relabel*, and the number of pushes of value $\geq \Delta$ is easily bounded by another application of part (a) with $\gamma = 1$. This shows the bound for nonsaturating pushes. As for saturating pushes, we define the concepts of small and terminal pushes as in the proof of Lemma 4.2(c) and argue as was done there. The number of small saturating pushes is bounded by m plus the number of nonsaturating pushes, there are $O(n^2\beta)$ terminal pushes, and the number of nonsmall nonterminal pushes is $O(n^2 \log U/l + nm/\beta + n^2\beta)$. □

The following lemma was essentially proved in [AOT89] (Lemma 4.2).

LEMMA 5.2. $\sharp$*waves* $= O(\min\{n^2, nl + \log U\})$.

Proof. We first show the $O(n^2)$ bound and then the $O(nl + \log U)$ bound.

For the $O(n^2)$ bound, observe first that at most $2n^2$ waves execute a relabeling (Lemma 3.6). On the other hand, a wave that does not execute at least one relabeling reduces Σ^* to zero and hence is either the last wave or is separated from the next wave by the addition to E^* of at least one edge. Thus there are at most $2n^2+m+1 = O(n^2)$ waves.

For the $O(nl + \log U)$ bound, consider any wave that is not the last in its phase. At the end of such a wave $\Sigma^* \geq n\Delta/l$. Also, no vertex has visible excess exceeding 2Δ (by Fact 5.1(b)), and every vertex with positive visible excess at the end of the wave was relabeled during the wave. Thus at least $n/(2l)$ relabelings occurred during the wave and hence the number of waves is bounded by $\lfloor \log U \rfloor + 1 + 2n^2/(n/(2l)) = O(nl + \log U)$. □

THEOREM 5.1. *A maximum flow in a network with n vertices, m edges, and integer capacities bounded by $U \geq 1$ can be computed deterministically using $O(q + \log U)$ flow operations and $O(q + \log U) + T_{ce}(n, m, q)$ time, where $q = O(n^{3/2}m^{1/2} + n^2(\log U)^{1/2})$.*

Proof. Replace U by $\max\{U, 2\}$ and choose l such that $l = \Theta(\sqrt{\log U})$ and such that the sequence $a_0, a_1, \ldots, a_{\lfloor \log U \rfloor}$ can be computed in $O(\log U)$ time, where $a_i = \lceil 2^i n/l \rceil$ for $i \geq 0$. We show below how to do this. Also take $\beta = \lfloor (m/n)^{1/2} \rfloor$ (as in the previous section) and note that β can be computed within the stated resources. As in the proof of Theorem 4.1, a component in the running time of $O(m \log n + \log U)$ accounts for initialization, maintenance of Δ, and multiplication of the capacities of all undirected edges by β. By the conditions placed on l above, each execution of the test between labels B and C can be carried out in constant time (maintain Σ^* explicitly). The total number of operations executed on the current-edge data structure as well as the remaining running time can be seen to be at most proportional to $\sharp$*pushes* $+$ $\sharp$*stack push relabels* $+ m$; in particular, note that the list J employed by *wave* can be constructed in $O(n)$ time by bucket sorting. Since $\sharp$*waves* $= O(\min\{n^2, nl + \log U\}) = O(n(\log U)^{1/2})$, Lemma 5.1 implies that $\sharp$*pushes*$+\sharp$*stack push relabels* $= O(n^{3/2}m^{1/2} + n^2(\log U)^{1/2})$, from which the desired result follows.

In the remainder of the proof, we show how to choose l to satisfy the conditions stated above. This material can be skipped in a first reading, and it is of little relevance to readers with no interest in the details of our model of computation.

What makes the problem nontrivial is the insistence of our model on a neat separation between the types *integer* and *flow value* and the very restricted operations applicable to values of type *flow value*. For example, the condition $U > n$ cannot be tested directly because of a type mismatch: U is of type *flow value*, while n is of type *integer*, and we have not provided for comparisons between values of different types. The available operations do, however, allow the computation of the *flow value* n from the *integer* n in $O(n)$ time (and, in fact, in $O(\log n)$ time), so that the test can be executed after all. We use this idea below.

Begin by computing $\lceil \log U \rceil = \min\{i \geq 1 : 2^i \geq U\}$ in $O(\log U)$ time using repeated doubling. Then determine $\lceil \sqrt{\log U} \rceil = \min\{i \geq 1 : \sum_{j=1}^{i}(2j-1) \geq \lceil \log U \rceil\}$ in $O(\sqrt{\log U})$ time. Finally compute $i_0 = \min\{i \in \mathbb{Z} : 2^i n \geq \lceil \sqrt{\log U} \rceil\}$ and take $l = 2^{i_0} n$. The numbers i_0 and $2^{|i_0|}$ can be found in $O(|i_0|+1) = O(\log(n+\log U))$ time via repeated doubling, starting from $\min\{n, \lceil \sqrt{\log U} \rceil\}$, and clearly $l = \Theta(\sqrt{\log U})$. Furthermore, $a_i = \lceil 2^{i-i_0} \rceil$ for $i \geq 0$. Hence a_0 can be computed in constant time, and a_i can be computed from a_{i-1} in constant time for all $i \geq 1$. □

6. Solutions to the current-edge problem.

In this section we describe two solutions to the current-edge problem, i.e., implementations of the current-edge data type, both of which are based on the fact that if an edge $(v, w) \in E^*$ is inadmissible at some time then it remains inadmissible until the next execution of *Relabel*(v) (cf. Lemma 3.4).

LEMMA 6.1. $T_{ce}(n, m, q) = O(nm + q) = O(n^3 + q)$.

Proof. Maintain for each vertex v a list L_v containing those edges (v, w) for which *Add edge*($\{v, w\}$) has been executed. If the functions r and h are represented by tables in the obvious way, *Init* can be executed in $O(m)$ time, while each of *Spush*, *Npush*, *Relabel*, and *Add edge* takes constant time. In order to implement the final operation *ce*, additionally maintain for each vertex v a pointer $z[v]$ into L_v which is initialized to point to the beginning of L_v and is reset to this position in each call of *Relabel*(v). Each call *ce*(v) advances $z[v]$ (possibly a distance of zero) until an admissible edge is encountered, or until the end of L_v is reached, in which case the value *nil* is returned. The correctness of this implementation follows from the fact cited above, which guarantees that no edge behind $z[v]$ is ever admissible; in particular, note that an edge (v, w) is always inadmissible at the time of its insertion in L_v. Since each pointer $z[v]$ makes at most $2n$ scans over its list L_v, the total time spent is $O(nm + q)$. □

Lemma 6.1 describes the standard solution to the current-edge problem introduced in [GT88] and also used in [AO89], [AOT89], and [CH95]. We now give a faster solution. First, identify V with the set $\{0, \ldots, n-1\}$ and extend r to a function from $V \times V$ to $\{0, 1\}$ by taking $r(v, w) = 0$ for $(v, w) \notin E$.

THEOREM 6.1. $T_{ce}(n, m, q) = O(n^3/\log n + q)$.

Proof. We represent the function h not only directly but also through an array $H : \{0, \ldots, 2n-1\} \times V \to \{0, 1\}$ such that for all integers k with $0 \leq k \leq 2n-1$ and all $v \in V$, $H[k, v] = 1$ if and only if $h(v) = k$. Then for all $(v, w) \in E$ with $h(v) > 0$, $r(v, w) \cdot H[h(v)-1, w] \neq 0$ iff the edge (v, w) is admissible. We combine this observation with the "four Russians' trick" (see [AHU74, §6.6]); i.e., we partition each row of the arrays r and H into *blocks* of size x and represent the x bits of each block by a single integer. Here x is a positive integer, which for simplicity we assume to be a divisor of n. More precisely, let $X = \{0, \ldots, 2^x - 1\}$. Instead of r and H, we maintain arrays $r' : V \times \{0, \ldots, n/x - 1\} \to X$ and $H' : \{0, \ldots, 2n-1\} \times \{0, \ldots, n/x - 1\} \to X$ defined as follows: for all $v \in V$ and all integers k and i with $0 \le k \le 2n - 1$ and $0 \le i \le n/x - 1$, let

$$r'[v,i] = \sum_{j=0}^{x-1} r(v, ix + j) \cdot 2^{x-1-j} \quad \text{and} \quad H'[k,i] = \sum_{j=0}^{x-1} H[k, ix + j] \cdot 2^{x-1-j}.$$

For $a \in X$ let $a^{(x-1)}, \ldots, a^{(0)}$ denote the individual bits of a, i.e., $a^{(x-1)}, \ldots, a^{(0)} \in \{0, 1\}$ and $\sum_{j=0}^{x-1} a^{(j)} \cdot 2^j = a$. For $a, b \in X$ let $a \wedge b$ be the bitwise AND of a and b; i.e., $a \wedge b = \sum_{j=0}^{x-1} (a^{(j)} \cdot b^{(j)}) \cdot 2^j$. Then for all $v \in V$ with $h(v) > 0$ and for all integers i with $0 \le i \le n/x - 1$, we have $r'[v,i] \wedge H'[h(v) - 1, i] \ne 0$ iff one of the edges in $\{(v, ix + j) : 0 \le j < x\}$ is admissible. This leads to the implementation of *ce* given below; the remaining operations are left to the reader. In order to understand the last line of the code, note that for each nonzero $a \in X$, $\lfloor \log a \rfloor$ is the position of the leftmost nonzero bit in a, the rightmost bit position counted as zero.

```
FUNCTION ce(v: vertex): edge;
  if $h(v) = 0$ then return nil fi;
  while $r'[v, z[v]] \wedge H'[h(v) - 1, z[v]] = 0$ and $z[v] < n/x - 1$
  do $z[v] := z[v] + 1$ od;
  if $r'[v, z[v]] \wedge H'[h(v) - 1, z[v]] = 0$
  then return nil
  else return $(v, z[v] \cdot x + x - 1 - \lfloor \log(r'[v, z[v]] \wedge H'[h(v) - 1, z[v]]) \rfloor)$ fi;
```

In the execution of any legal sequence of q operations following *Init*, the total number of changes to $z[v]$ is $O(n^2/x)$ for arbitrary $v \in V$. Hence such a sequence can be executed in $O(n^3/x + q)$ time, provided that the operations of testing and setting individual bits of numbers in X and of computing $\lfloor \log a \rfloor$ and $a \wedge b$ for arbitrary $a, b \in X$ take constant time.

For $x = \lfloor \log n \rfloor$, tables implementing the operations $a \mapsto \lfloor \log a \rfloor$ and $(a, b) \mapsto a \wedge b$ for $a, b \in X$ can be constructed in $O(n^2)$ time, and individual bits of numbers in X can be inspected and modified in constant time via appropriate multiplications and integer divisions by powers of two. This completes the proof of Theorem 6.1. □

Remark. On many real computers, the operation of bitwise AND is built in, i.e., takes constant time. It is easy to improve Theorem 6.1 for such a machine with a nonstandard word length of $x = \omega(\log n)$ bits. Although the remaining bit-level operations discussed above may not be available at unit cost, they can trivially be executed in $O(x)$ time; hence on a machine with a word length of x bits and unit-time bitwise AND, $T_{ce}(n, m, q) = O(n^3/x + qx + n^2)$, where the term n^2 accounts for the cost of initialization.

7. The incremental strongly polynomial algorithm.

In addition to the data structures of the generic algorithm, the incremental strongly polynomial algorithm uses, as do several previous algorithms, an edge-weighted directed graph $F = (V, E_F, val)$, where $E_F \subseteq E^*$ and *val* is a function from E_F to $\mathbb{R}$. F at all times is a directed forest, i.e., an acyclic directed graph with maximum outdegree at most one, and $val(e) = rescap(e)$ for all $e \in E_F$. A vertex $v \in V$ is called a *root* exactly if its outdegree in F is zero. The following operations are applied to F:

InitF;
Sets $E_F := \emptyset$;

Find value(e);
Precondition: $e \in E_F$.
Returns *val*(e);

Find root(v);
Precondition: $v \in V$.
Returns the root of the tree in F containing v;

Find min(v);
Precondition: $v \in V$ and v is not a root.
Returns an edge e of minimal value *val*(e) on the maximal path in F starting at v; in the case of ties the last such edge is returned;

Add value(v, c);
Precondition: $v \in V$ and $c \in \mathbb{R}$.
Replaces *val*(e) by *val*$(e) + c$ for each edge e on the maximal path in F starting at v;

Link(e, c);
Precondition: $e \in E^*$, $c \in \mathbb{R}$, and $(V, E_F \cup \{e\})$ is a directed forest.
Replaces E_F by $E_F \cup \{e\}$ and sets *val*$(e) := c$;

Cut(e);
Precondition: $e \in E_F$.
Replaces E_F by $E_F \backslash \{e\}$;

The *dynamic trees* data structure of Sleator and Tarjan [ST85] supports the seven operations defined above in $O(\log n)$ amortized time each; i.e., a sequence of q operations on F, starting with *InitF*, can be executed in $O(q \log n)$ time.

The preflow f is represented in one of two ways: for $e \in E^*$, while $e \notin E_F$ and $rev(e) \notin E_F$, $f(e)$ is stored directly as $g[e]$, where $g : E \rightarrow \mathbb{R}$ is an array. While $e \in E_F$, $f(e)$ is given implicitly as $cap(e) - val(e)$ and $f(rev(e))$ as $-f(e)$. Accordingly, we redefine the basic procedure *setflow* and incorporate the conventions for the representation of f into new versions of *Link* and *Cut*.

```
PROCEDURE setflow(e: edge; c: real);
  $g[e] := c$; $g[rev(e)] := -c$;

PROCEDURE link(e: edge);
  Link(e, rescap(e));

PROCEDURE cut(e: edge);
  setflow(e, cap(e) − Find value(e));
  Cut(e);
```

The procedure *tree push* defined below works as follows: a call *tree push*(v) first inserts an eligible edge with tail v into E_F if v is a root, and then determines the minimal residual capacity c of any edge on the maximal path in E_F starting at v. It finally increases the flow along that path by $\min\{c, excess^*(v)\}$ and deletes all edges from E_F that become saturated.

```
PROCEDURE tree push(v: vertex);
  if Find root(v) = v (* v is a root *) then link(ce(v)) fi;
  c := Find value(Find min(v));
  Add value(v, − min{c, excess*(v)});
  while Find root(v) ≠ v and Find value(Find min(v)) = 0
  do $cut(Find\ min(v))$ od;
```

We finally extend the routine *relabel* and give the main program.

```
PROCEDURE $relabel(v\colon\ vertex)$;
  for all $u \in V$ with $(u, v) \in E_F$ do $cut(u, v)$ od;
  $d(v) := d(v) + 1$;
```

```
INCREMENTAL STRONGLY POLYNOMIAL ALGORITHM:
  generic initialize;
  InitF;
  $L :=$ list of the undirected edges in $\overline{E}$ ordered by decreasing capacities;
  while $L \neq \emptyset$
  do
    $\Delta := cap(first(L))$;        (* $\Delta$ is used only by the analysis *)
    $add\ edge(pop(L))$;
    while $\max\{excess^*(v) : v \in V^+\} > 0$
    do
      Choose $v \in V^+$ with $excess^*(v) > 0$;
      if $ce(v) = nil$
      then $relabel(v)$
      else $tree\ push(v)$;
      fi;
    od;
  od.
```

The algorithm uses the current-edge data type as follows: *generic initialize* calls *Init*, *relabel*(v) calls *Relabel*(v), *add edge*$(\{v, w\})$ calls *Add edge*$(\{v, w\})$, *link*(e) calls *Npush*(e), and each call *cut*(e) within *tree push* calls *Spush*(e). As will be seen below, the latter conventions regarding calls of *Npush* and *Spush* guarantee the invariant that each edge in E^* is eligible iff it is admissible, which in turn ensures the correctness of the values returned by calls of *ce*. The argument given in §3 for the equivalence of eligibility and admissibility no longer suffices. While clearly $h(v) = d(v)$ continues to hold for all $v \in V$, the fact that an edge $(v, w) \in E^*$ is residual iff $r(v, w) = 1$ is demonstrated in part (b) of Lemma 7.1.

LEMMA 7.1. *At all times of the execution after the initialization and except during calls of add edge, tree push, and relabel, the following invariants hold:*

(a) *every edge in E_F is eligible*;

(b) *for all $(v, w) \in E^*$, (v, w) is residual iff $r(v, w) = 1$.*

Proof. (a) and (b) hold vacuously immediately after the initialization (at which point $E_F = E^* = \emptyset$). Let us therefore assume, by way of induction, that they hold prior to a call of *add edge*, *tree push*, or *relabel*. We show that they hold after the call.

Invariant (a) can only be violated by a step that inserts an edge in E_F or that makes an edge in E_F ineligible. When an edge is inserted in E_F (in the call *link*$(ce(v))$), it was returned by *ce* immediately prior to the operation and hence is admissible at the time of the insertion. By invariant (b), it is also eligible at that time, so that invariant (a) is preserved. Note also that since traversing an eligible edge is always accompanied by a decrease in d value, the new edge does not form a cycle with the edges already in E_F; i.e., the precondition of the call of *Link* is satisfied. An edge in E_F becomes ineligible either because it is saturated, in which case it is removed from E_F in a call of *tree push*, or because of a relabeling, in which case it is removed from E_F in a call of *relabel.* In either case, since the edge no longer belongs to E_F, invariant (a) is preserved.

We now turn to invariant (b). Recall first from §3 that a call *Add edge*$(\{v,w\})$ initializes $r(v,w)$ and $r(w,v)$ correctly, i.e., in accordance with the invariant. When an edge (v,w) is inserted in E_F, a call *Npush*(v,w) sets $r(v,w)=1$, and $r(v,w)=1$ remains true until the call *Spush*(v,w), executed when (v,w) is saturated and removed from E_F (if this ever happens), which sets $r(v,w)=0$. Thus, by invariant (a), invariant (b) holds for all edges in E_F. When an edge (w,v) is inserted in E_F in a call of *tree push*, $r(v,w)$ is also set to 1. Furthermore, by invariant (a), the value of c computed by the call of *tree push* is strictly positive, so that a positive amount of flow is sent over (w,v) in the same call, making (v,w) residual if it were not so already. It is easy to see that as long as (w,v) remains in E_F, $r(v,w)$ remains equal to 1, and (v,w) remains residual. Invariant (b) therefore holds also for edges (v,w) such that (w,v) is in E_F. Since flow is pushed only over edges in E_F, while only insertions into and deletions from E_F change values of r, invariant (b) clearly holds for the remaining edges as well. □

As an instance of the incremental generic algorithm, the incremental strongly polynomial algorithm is partially correct. The following fact is obvious.

Fact 7.1. At all times during the execution, $excess^*(v) \le \Delta$ for all vertices $v \in V^+$.

Define a *PTR* (*premature target relabeling*) *event* on an edge e to be a relabeling of the head of e while e is in E_F, and denote by $\sharp ptr$ the total number of PTR events during the execution. PTR events were introduced in [CH89], although with a somewhat different meaning. Their number depends on the exact edges returned by calls of *ce*; this dependence will be discussed in §8. A PTR event on an undirected edge $\{v,w\}$ is a PTR event on one of the edges (v,w) or (w,v). Denote by $\sharp$*sat cuts* the number of calls of *cut* within *tree push*, by $\sharp$*cuts* the sum of $\sharp$*sat cuts* and $\sharp ptr$ (observe that $\sharp ptr$ is the number of calls of *cut* within *relabel*), by $\sharp$*tree pushes* the number of calls of *tree push*, and by $\sharp$*links* the number of calls of *link*. A call of *cut* or *link* is also called a *cut* or a *link*, respectively.

The analysis of our strongly polynomial algorithm centers around the following ideas. We first show that the running time is determined by the search for current edges, $\sharp$*tree pushes*, and $\sharp$*cuts* and then relate $\sharp$*tree pushes* to $\sharp$*cuts*. In order to bound $\sharp$*sat cuts*, we use a potential function similar to the one used in the proof of Lemma 4.2, and in order to bound $\sharp ptr$, we use the analysis of [CH89].

LEMMA 7.2. *The algorithm uses* $O(q\log n)$ *flow operations and* $O((\sharp\textit{tree pushes}+\sharp\textit{cuts})\cdot\log n+n^2+m\log n)+T_{ce}(n,m,q)$ *time, where* $q=O(\sharp\textit{tree pushes}+\sharp\textit{cuts}+n^2)$.

Proof. It takes time $O(m\log n)$ to sort the undirected edges by capacity. If we maintain for each vertex v the set of edges in E_F with head v, then a relabeling takes $O(1)$ time plus $O(\log n)$ time for each cut caused by the relabeling. A call of *tree push* takes $O(\log n)$ time plus $O(\log n)$ time for each cut caused by *tree push.* Finally, the time spent on the current-edge task is $T_{ce}(n,m,q)$, where $q=O(\sharp\textit{tree pushes}+\sharp\textit{cuts}+n^2)$, since the number of calls of *ce* is bounded by the number of relabelings plus twice the number of calls of *tree push*, the numbers of calls of *Npush* and *Spush* are bounded by $\sharp$*tree pushes* and $\sharp$*cuts*, respectively, and there are $O(n^2)$ calls of *Add edge* and *Relabel.* □

LEMMA 7.3.

(a) $\sharp\textit{tree pushes}=O(\sharp\textit{links}+\sharp\textit{sat cuts}+m)$;

(b) $\sharp\textit{links}\le\sharp\textit{cuts}+n$.

Proof. (a) We use a potential function Φ defined as the number of nonroot vertices with positive visible excess. When *tree push*(v) is called, we have $\mathit{excess}^*(v) > 0$. If a call *tree push*(v) performs neither a link nor a cut, then v was not a root before the call and $\mathit{excess}^*(v) = 0$ after the call; i.e., Φ is reduced by one. No call of *tree push* increases Φ by more than one, the addition of an undirected edge increases Φ by at most two, and a relabeling does not change Φ. Hence the total increase in Φ is $O(\sharp \mathit{links} + \sharp \mathit{sat\ cuts} + m)$, $\Phi = 0$ initially, and $\Phi \geq 0$ always, and, with the exception of at most $\sharp \mathit{links} + \sharp \mathit{sat\ cuts}$ calls, every call of *tree push* decreases Φ by one.

(b) Since F is a forest at all times during the execution, it never contains more than $n - 1$ edges. □

LEMMA 7.4. *$\sharp \mathit{sat\ cuts} = O(n^{3/2}m^{1/2} + \sharp \mathit{ptr})$.*

Proof. Define a *push bundle* for an edge $e \in E^*$ to be the sequence of all regular pushes on e in a maximal period of time in which e belongs to E_F. A push bundle for an undirected edge $\{v, w\}$ is a push bundle for one of the edges (v, w) or (w, v). The number of push bundles clearly bounds $\sharp \mathit{sat\ cuts}$. A push bundle for an edge $e = (v, w) \in E^*$ is called *complete* if its pushes increase $f(e)$ by $\mathit{cap}(\{v, w\})$, i.e., from $-\mathit{cap}(w, v)$ to $\mathit{cap}(v, w)$, and *incomplete* otherwise.

CLAIM 1. *The number of incomplete push bundles is $O(m + \sharp \mathit{ptr})$.*

Proof. A maximal period of time in which an edge e belongs to E_F is terminated by a PTR event on e, by a saturating push on e, or by the end of the execution. Hence an incomplete push bundle for an undirected edge $\{v, w\}$ that is neither the first nor the last push bundle for $\{v, w\}$ is either immediately preceded or immediately followed by a PTR event on $\{v, w\}$.

CLAIM 2. *The number of complete push bundles is $O(n^{3/2}m^{1/2})$.*

Proof. Define the *level* of a push on an edge $(u, v) \in E^*$ to be the value of $d(u)$ at the time of the push. Two pushes on a fixed edge (u, v) have the same level if and only if they belong to the same push bundle. Hence for all $(u, v) \in E$ and all integers k with $1 \leq k \leq 2n - 1$, we can denote by $\langle u, v, k\rangle$ the push bundle for (u, v) (if any) whose pushes are of level k. Let β be a positive integer, to be chosen below, and call a push bundle $\langle u, v, k\rangle$ *terminal* if it is followed by fewer than β push bundles of the form $\langle u, w, k\rangle$, where $(u, w) \in E$. Clearly, there are at most $2n^2\beta$ terminal push bundles. In order to count the number of nonterminal push bundles, we use a potential argument similar to those used in the proofs of Lemmas 4.2 and 5.1.

Consider the potential function

$$\Phi = \sum_{v \in V^+} \frac{\mathit{excess}^*(v)}{\Delta} \cdot d_\beta(v),$$

where d_β denotes the β-fooling height introduced in §4. $\Phi = 0$ initially and $\Phi \geq 0$ always (by Lemma 3.1 and property (F2) of β-fooling height), Φ does not increase due to changes of Δ (since $\mathit{excess}^*(v) \leq 0$ for all $v \in V^+$ at each change of Δ), Φ does not increase due to regular pushes (by property (F3)), the total increase due to relabelings is at most $2n^2$ (by Fact 7.1 and property (F5)), and the increase due to the addition of an undirected edge is at most n/β (by property (F1)). The total decrease of Φ is therefore bounded by $nm/\beta + 2n^2$. Finally, note that the pushes in a complete nonterminal push bundle decrease Φ by at least one. This can be seen as follows. Consider a complete nonterminal push bundle $\langle u, v, k\rangle$. Since $\langle u, v, k\rangle$ is nonterminal, it is followed by β bundles $\langle u, w_1, k\rangle, \ldots, \langle u, w_\beta, k\rangle$. Lemma 3.4 now implies that the edges $(u, w_1), \ldots, (u, w_\beta)$ are eligible whenever a push in the bundle $\langle u, v, k\rangle$ occurs. Thus, by property (F4), $d_\beta(u) > d_\beta(v)$ at the time of each such push. Also, the total value of the pushes in the push bundle is $cap(\{u,v\})$ and $\Delta \leq cap(\{u,v\})$ whenever a push in the bundle occurs since the undirected edges are added in the order of decreasing capacities. Summing up, the total number of complete push bundles is $O(nm/\beta + n^2\beta)$. Claim 2 follows with $\beta = \lfloor (m/n)^{1/2} \rfloor$, and this ends the proof of Lemma 7.4. □

LEMMA 7.5. *The algorithm uses $O(q \log n)$ flow operations and $O(q \log n) + T_{ce}(n,m,q)$ time, where $q = O(n^{3/2}m^{1/2} + \sharp ptr)$.*

Proof. Combine Lemmas 7.2, 7.3, and 7.4. □

8. The extended current-edge problem and PTR events.

In the definition of the current-edge data type in §3, we allowed a call $ce(v)$ to return an arbitrary admissible edge (if any) with tail v. Given so much freedom, however, an adversary might be able to "score" a high number of PTR events, leading to a bad running time. Our defense against the adversary will be randomness: we force the choice among several admissible edges to be made according to a fixed but random ordering; then we can prove that the number of PTR events is usually much lower than the naive upper bound. In this section, we adapt the specification of the current-edge data type and extend the results of §6 to the more restrictive definition of *ce*, review and slightly extend the bounds on the number of PTR events shown in [CH89], and finally prove our main theorem.

For every finite set A, denote by Perm(A) the set of all permutations of A, i.e., of all bijections from $\{0,\dots,|A|-1\}$ to A. As in §6, identify V with the set $\{0,\dots,n-1\}$. The *extended current-edge data type* is initialized with n permutations $\xi_0, \xi_1, \dots, \xi_{n-1}$ of V. Its task is to maintain two functions $r : E \to \{0,1\}$ and $h : V \to \{0,\dots,2n-1\}$ under the operations *Init*, *Spush*, *Npush*, *Relabel*, *Add edge*, and *ce*. The operations *Spush*, *Npush*, *Relabel*, and *Add edge* are defined as in §3, and *Init* and *ce* are redefined as follows:

Init$(\xi_0,\dots,\xi_{n-1})$;
Precondition: $\xi_0,\dots,\xi_{n-1}$ are permutations of V.
Records $\xi_0,\dots,\xi_{n-1}$ and sets $h(v) := 0$ for $v \in V\backslash\{s\}$, $h(s) := n$, and $r(v,w) := 0$ for all $(v,w) \in E$.

ce(v);
Precondition: $v \in V$.
Returns the first admissible edge with tail v in the order induced by ξ_v if $E(v) \neq \emptyset$, *nil* otherwise.
If $E(v) \neq \emptyset$, the first admissible edge with tail v in the order induced by ξ_v is $(v, \xi_v(i_0))$, where $i_0 = \min\{i : 0 \leq i \leq n-1 \text{ and } (v,\xi_v(i)) \in E(v)\}$.

For $q \in \mathbb{N}$, denote by $T'_{ce}(n,m,q)$ the time needed to execute any legal sequence of one *Init* operation followed by q *Spush*, *Npush*, *Relabel*, *Add edge*, and *ce* operations of the extended current-edge data type.

LEMMA 8.1. $T'_{ce}(n,m,q) = O(nm + q) = O(n^3 + q)$.

Proof. The proof is identical to that of Lemma 6.1, except that the list L_v is kept sorted according to the order induced by ξ_v for all $v \in V$. This makes *Add edge* operations more time-consuming. Since there are at most m calls of *Add edge*, however, each of which can be executed in $O(n)$ time, the total time is still $O(nm + q)$. □

We now extend the faster solution of §6, but only for a restricted class of permutations $\xi_0,\dots,\xi_{n-1}$. Let $x = \lfloor \log n \rfloor$, which, as in §6, we assume to be a divisor of n. Also, take $M = \{0, \ldots, n/x - 1\}$ and let $B_i = \{ix, ix+1, \ldots, (i+1)x - 1\}$, for $i = 0, \ldots, n/x - 1$. A permutation of M is called a *block permutation.* For every block permutation $\Xi \in \mathrm{Perm}(M)$, define the *induced block-preserving permutation* as the permutation $\xi \in \mathrm{Perm}(V)$ obtained by first arranging the blocks according to Ξ and then replacing each block by the sorted sequence of its elements (i.e., for $v \in B_i$ and $w \in B_j$, $\xi^{-1}(v) < \xi^{-1}(w) \Longleftrightarrow (\Xi^{-1}(i) < \Xi^{-1}(j)$ or $(i = j$ and $v < w)))$.

LEMMA 8.2. *For all $q \in \mathbb{N}$ and for n arbitrary block permutations $\Xi_0, \ldots, \Xi_{n-1} \in \mathrm{Perm}(M)$ with induced block-preserving permutations $\xi_0, \ldots, \xi_{n-1} \in \mathrm{Perm}(V)$, the operation Init$(\xi_0, \ldots, \xi_{n-1})$ and any legal sequence of q Spush, Npush, Relabel, Add edge, and ce operations following it can be executed in $O(n^3/\log n + q)$ time.*

Proof. The proof of Theorem 6.1 carries over with only two minor changes: a relabeling of a vertex v resets $z[v]$ to $\Xi_v(0)$ instead of to 0, and in the implementation of *ce* the pointer $z[v]$ steps through the blocks in the order given by Ξ_v instead of in increasing order; i.e., lines 3 and 4 of the code of *ce* are replaced by

```
while $r'[v, z[v]] \wedge H'[h(v) - 1, z[v]] = 0$ and $\Xi_v^{-1}(z[v]) < n/x - 1$
do $z[v] := \Xi_v(\Xi_v^{-1}(z[v]) + 1)$ od;    □
```

The incremental strongly polynomial algorithm uses the extended current-edge data type essentially as described in §7 (in the paragraph preceding Lemma 7.1). The only modification is that *generic initialize* now chooses n permutations $\xi_0, \ldots, \xi_{n-1}$ of V and calls *Init*$(\xi_0, \ldots, \xi_{n-1})$. In this situation, we say that the algorithm is executed with the adjacency lists ordered according to $\xi_0, \ldots, \xi_{n-1}$.

We now turn to the discussion of PTR events. We need the following definitions. 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 $\mu(0), \ldots, \mu(|A| - 1)$ and $\sigma(0), \ldots, \sigma(|B| - 1)$. Given l permutations $\mu_0, \ldots, \mu_{l-1}$ of subsets of a finite set A, for some $l \in \mathbb{N}$, let $\Lambda(\mu_0, \ldots, \mu_{l-1}) = \max_{\sigma \in \mathrm{Perm}(A)} \sum_{i=0}^{l-1} \lambda(\mu_i, \sigma)$; note that this quantity does not depend on A. We call $\Lambda(\mu_0, \ldots, \mu_{l-1})$ the *external coascent* of $\mu_0, \ldots, \mu_{l-1}$.

For all $u \in V$, denote by Γ_u the set of neighbors of u; i.e., $\Gamma_0 = \{v \in V : (u, v) \in E\}$. For $\xi \in \mathrm{Perm}(V)$ and $u \in V$, call $\mu \in \mathrm{Perm}(\Gamma_u)$ the *restriction* of ξ to Γ_u if the vertices in Γ_u are ordered identically by μ and by ξ, i.e., if $\mu^{-1}(v) < \mu^{-1}(w) \Longleftrightarrow \xi^{-1}(v) < \xi^{-1}(w)$ for all v, $w \in \Gamma_u$.

LEMMA 8.3 (*see* [CH89]). *Let $\xi_0, \ldots, \xi_{n-1} \in \mathrm{Perm}(V)$. If the strongly polynomial algorithm is executed with the adjacency lists ordered according to $\xi_0, \ldots, \xi_{n-1}$, then $\sharp ptr \le 2n \cdot \Lambda(\mu_0, \ldots, \mu_{n-1})$, where μ_v is the restriction of ξ_v to Γ_v for all $v \in V$.*

Proof. 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 $k = 0, \ldots, 2n - 2$ and for 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 an execution of the algorithm with the adjacency lists ordered according to $\xi_0, \ldots, \xi_{n-1}$ that relabels according to $\sigma_0, \ldots, \sigma_{2n-2}$ and fix $v \in V$ and $k \in \{0, \ldots, 2n - 2\}$. We will count the number $\sharp ptr_{v,k}$ of PTR events on edges with tail v while $d(v) = k$. Suppose that $w_1, \ldots, w_l$ are vertices in V such that the algorithm incurs PTR events on the edges $(v, w_1), \dots, (v, w_l)$, in that order, while $d(v) = k$. 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 $\mu_v(1), \dots, \mu_v(|\Gamma_v|)$ and $\sigma_k(1), \dots, \sigma_k(n)$ have a (not necessarily contiguous) subsequence of length l, namely $w_1, \dots, w_l$. Thus $\sharp ptr_{v,k} \le \lambda(\mu_v, \sigma_k)$. Summing over all values of v and k yields

$$\sharp ptr \le \sum_{k=0}^{2n-2} \sum_{v \in V} \lambda(\mu_v, \sigma_k) \le \sum_{k=0}^{2n-2} \Lambda(\mu_0, \dots, \mu_{n-1}) \le 2n \cdot \Lambda(\mu_0, \dots, \mu_{n-1}). \qquad \square$$

As is clear from Lemma 8.3, our next task is to analyze $\Lambda(\mu_0, \dots, \mu_{n-1})$, where $\mu_0, \dots, \mu_{n-1}$ are obtained in various different ways.

LEMMA 8.4.

(a) *For all $v \in V$, let μ_v be a permutation of Γ_v. Then $\Lambda(\mu_0, \dots, \mu_{n-1}) \le m$.*

(b) (*See* [Al90].) *For every two integers n and h with $1 \le h \le n$ and every set W with $|W| = h$, n permutations $\mu_0, \dots, \mu_{n-1}$ of W with $\Lambda(\mu_0, \dots, \mu_{n-1}) = O(nh^{2/3})$ can be constructed in $O(nh)$ time.*

(c) *Suppose that μ_v is drawn randomly from the uniform distribution over $\mathrm{Perm}(\Gamma_v)$ for all $v \in V$ and that $\mu_0, \dots, \mu_{n-1}$ are independent. Take $\zeta = \log(2 + n(\log n)^2/m)$. Then for some $\theta = \theta(n, m)$ with $\theta = O(\sqrt{nm} + n \log n/\zeta)$ and for all $r \ge 0$,*

$$\Pr(\Lambda(\mu_0, \dots, \mu_{n-1}) \ge \theta + r) \le 2^{-r}.$$

Remark. The proof of part (c) is based on the proofs of Lemma 10 in [CH89] and of Lemma 6.3 in [CH95]. For $m = o(n(\log n)^2)$, it strengthens those lemmas.

Proof.

(a) This is obvious since $|\Gamma_0| + \cdots + |\Gamma_{n-1}| = m$.

(b) This is Theorem 2 in [Al90].

(c) Recall that $\Lambda(\mu_0, \dots, \mu_{n-1}) = \max_{\sigma \in \mathrm{Perm}(V)} \phi(\sigma)$, where $\phi(\sigma) = \sum_{v=0}^{n-1} \lambda(\mu_v, \sigma)$. We will show the probability that $\phi(\sigma)$ is large to be very small for each fixed $\sigma \in \mathrm{Perm}(V)$. Multiplying that probability by the number of choices for σ, i.e., by $n!$, we obtain an upper bound on the probability that $\Lambda(\mu_0, \dots, \mu_{n-1})$ is large.

Hence let $\sigma \in \mathrm{Perm}(V)$ be arbitrary but fixed. For all $v \in V$, let $\Lambda_v = \lambda(\mu_v, \sigma)$ and take $S = \phi(\sigma) = \sum_{v=0}^{n-1} \Lambda_v$, the quantity of interest. For all $v \in V$, let d_v be the degree of v; i.e., $d_v = |\Gamma_v|$.

For arbitrary integers d and k with $0 \le k \le d \le n$, the number of permutations μ of an arbitrary subset of V of cardinality d with $\lambda(\mu, \sigma) \ge k$ is at most $\binom{d}{k}^2 (d-k)!$. To see this, note that if $\lambda(\mu, \sigma) \ge k$, then the elements of a (not necessarily contiguous) subsequence of $\mu(0), \dots, \mu(d-1)$ of length k appear in the same order in the sequence $\sigma(0), \dots, \sigma(n-1)$. The elements of the subsequence can be chosen in $\binom{d}{k}$ ways, and the positions in which they appear in $\mu(0), \dots, \mu(d-1)$ can also be chosen in $\binom{d}{k}$ ways, while the remainder of the sequence $\mu(0), \dots, \mu(d-1)$ can be chosen in $(d-k)!$ ways. It follows that for all $v \in V$ and all integers k with $1 \le k \le d_v$,

$$\Pr(\Lambda_v \ge k) \le \frac{\binom{d_v}{k}^2 (d_v - k)!}{d_v!} \le \frac{d_v^k}{(k!)^2} \le \left(\frac{e^2 d_v}{k^2}\right)^k,$$

where in the last step we used (a very crude) Stirling's approximation $k! \ge (k/e)^k$.

It can be seen that Λ_v is unlikely to exceed $\sqrt{d_v}$ by very much. Applying the Cauchy–Schwarz inequality $|u \cdot v| \le |u||v|$ to the vectors $u = (1, \ldots, 1)$ and $v = (\sqrt{d_0}, \ldots, \sqrt{d_{n-1}})$, we obtain

$$\sum_{v=0}^{n-1} \sqrt{d_v} \le \sqrt{n} \cdot \sqrt{\sum_{v=0}^{n-1} d_v} = \sqrt{nm}.$$

$S = \sum_{v=0}^{n-1} \Lambda_v$ is hence unlikely to exceed $\sqrt{nm}$ by very much. In order to obtain precise bounds, we use a method based on the moment-generating functions of $\Lambda_0, \ldots, \Lambda_{n-1}$ and akin to the usual proof of the well-known Chernoff bounds (see, e.g., [CLR90] or [HR90]).

First, observe that for arbitrary real numbers θ, r, and t with $t \ge 1$,

$$\Pr(S \ge \theta + r) = e^{-t(\theta+r)} e^{t(\theta+r)} \Pr(e^{tS} \ge e^{t(\theta+r)}) \le e^{-t(\theta+r)} E(e^{tS}),$$

where the simple Markov inequality was used in the last step. Second, since $\mu_0, \ldots, \mu_{n-1}$ and hence also $e^{t\Lambda_0}, \ldots, e^{t\Lambda_{n-1}}$ are independent,

$$E(e^{tS}) = E\left(e^{\sum_{v=0}^{n-1}(t\Lambda_v)}\right) = E\left(\prod_{v=0}^{n-1} e^{t\Lambda_v}\right) = \prod_{v=0}^{n-1} E(e^{t\Lambda_v}).$$

We next bound the quantities $E(e^{t\Lambda_v})$. Let $v \in V$ and let $a_v \ge 0$ be an arbitrary integer. Then

$$E(e^{t\Lambda_v}) = \sum_{k=0}^{\infty} e^{tk} \Pr(\Lambda_v = k) \le \sum_{k=0}^{a_v} e^{tk} \Pr(\Lambda_v = k) + \sum_{k=a_v+1}^{\infty} e^{tk} \Pr(\Lambda_v \ge k)$$
$$\le e^{ta_v} \sum_{k=0}^{a_v} \Pr(\Lambda_v = k) + \sum_{k=a_v+1}^{\infty} e^{tk} \left(\frac{e^2 d_v}{k^2}\right)^k \le e^{ta_v} + \sum_{k=a_v+1}^{\infty} \left(\frac{e^{t+2} d_v}{k^2}\right)^k.$$

Choose a_v to make $\frac{e^{t+2} d_v}{k^2} \le \frac{1}{2}$ for $k \ge a_v + 1$; i.e., take $a_v = \lfloor \sqrt{2e^{t+2} d_v} \rfloor$. Then

$$E(e^{t\Lambda_v}) \le e^{ta_v} + \sum_{k=a_v+1}^{\infty} 2^{-k} = e^{ta_v} + 2^{-a_v} \le 2e^{t\sqrt{2e^{t+2}d_v}} \le 2e^{te^{t+3}\sqrt{d_v}}.$$

Putting everything together yields

$$\Pr(S \ge \theta + r) \le e^{-t(\theta+r)} E(e^{tS}) = e^{-t(\theta+r)} \prod_{v=0}^{n-1} E(e^{t\Lambda_v}) \le e^{-t(\theta+r)} \prod_{v=0}^{n-1} (2e^{te^{t+3}\sqrt{d_v}})$$
$$= e^{-t(\theta+r)} \cdot 2^n e^{te^{t+3} \sum_{v=0}^{n-1} \sqrt{d_v}} \le 2^n e^{t(e^{t+3}\sqrt{nm} - (\theta+r))}.$$

Recalling that σ can be chosen in $n!$ ways, we find

$$\Pr(\Lambda(\mu_0, \ldots, \mu_{n-1}) \ge \theta + r) \le n! \cdot 2^n e^{te^{t+3}\sqrt{nm} - t\theta - tr}$$
$$\le 2^{2n \log n} e^{te^{t+3}\sqrt{nm} - t\theta} e^{-tr} \le e^{2n \log n + te^{t+3}\sqrt{nm} - t\theta} \cdot 2^{-r}.$$

Choose θ so as to make $2n \log n + te^{t+3}\sqrt{nm} - t\theta = 0$; i.e., take

$$\theta = \frac{2n \log n + te^{t+3}\sqrt{nm}}{t}.$$

Then $\Pr(\Lambda(\mu_0, \ldots, \mu_{n-1}) \geq \theta + r) \leq 2^{-r}$, as desired. All that remains is to show that for all combinations of n and m, it is possible to choose $t \geq 1$ such that $\theta = O(\sqrt{nm} + n \log n/\zeta)$. Consider two cases:

Case 1: $e^4\sqrt{nm} > n \log n$. In this case, take $t = 1$ and observe that $\theta = O(\sqrt{nm})$, as required. This is essentially the analysis of [CH89].

Case 2: $e^4\sqrt{nm} \leq n \log n$. Now choose $t \geq 1$ to make $te^{t+3}\sqrt{nm} = n \log n$; i.e.,

$$te^{t+3} = \sqrt{\frac{n(\log n)^2}{m}}.$$

This is clearly possible, and $t = \Omega(\zeta)$. But then $\theta = O(n \log n/\zeta)$. □

THEOREM 8.1. *A maximum flow in a network with* n *vertices and* m *edges can be computed with the following bounds on flow operations and time:*

(a) *deterministically using* $O(nm \log n)$ *flow operations and* $O(nm \log n)$ *time*;

(b) *deterministically using* $O(q \log n)$ *flow operations and* $O(nm + q \log n)$ *time, where* $q = n^{8/3}$.

(c) *probabilistically using* $O(\alpha q \log n)$ *flow operations and* $O(nm + \alpha q \log n)$ *time with probability at least* $1 - 2^{-\alpha\sqrt{nm}}$, *for arbitrary* $\alpha \geq 1$, *where* $q = n^{3/2}m^{1/2} + n^2 \log n/\log(2 + n(\log n)^2/m)$.

Remark. The bounds of part (a) were previously obtained by [ST83]. The time bound of part (b) was previously obtained by [Al90], although with a weaker bound on the number of flow operations. For $m = \Omega(n(\log n)^2)$, the time bound of part (c) was previously obtained by [Ta89] and [CH95], although with a weaker bound on the number of flow operations. For $m = o(n(\log n)^2)$, the result is new.

Proof.

(a) Combine Lemmas 7.5, 8.3, 8.4(a), and 6.1.

(b) Combine Lemmas 7.5, 8.3, 8.4(b) (used with $h = n$), and 8.1. (Note that with $T_{ce}(n, m, q)$ replaced by $T'_{ce}(n, m, q)$, Lemma 7.5 holds for the modified algorithm that works with the extended current-edge data type.)

(c) Initialize the current-edge data structure with n independent random permutations $\xi_0, \ldots, \xi_{n-1}$ of V. Since random permutations can be computed in linear time (see, e.g., [Se77]), this can be done in $O(n^2)$ time. Taking $r = \alpha\sqrt{nm}$ in Lemma 8.4(c) and using also Lemma 8.3, conclude that except with probability at most $2^{-\alpha\sqrt{nm}}$, we have $\sharp ptr = O(\alpha q)$. The claim now follows from Lemmas 7.5 and 8.1. □

Remark. If, as in [CH89], a new random permutation ξ_v of Γ_v is computed at each relabeling of v for all $v \in V$ then the failure probability of part (c) can be reduced even further to $2^{-\alpha q}$.

Remark. Following [AOT89], we can combine the incremental wave scaling algorithm of §5 with the use of dynamic trees. Since this requires few new ideas, we omit the details and only state the following result: for every $\alpha \geq 1$, a maximum flow in a network with n vertices, m edges, and integer capacities bounded by $U \geq 1$ can be computed using $O(\alpha q \log(2 + n \log U/m) + \log U)$ flow operations and $O(nm + \alpha q \log(2 + n \log U/m) + \log U)$ time with probability at least $1 - 2^{-\alpha\sqrt{nm}}$, where $q = n^{3/2}m^{1/2} + n^2 \log n/\log(2 + n(\log n)^2/m)$.

In order to use the faster solution to the extended current-edge problem provided by Lemma 8.2, we first need to demonstrate that random block-preserving permutations are almost as "good" as unrestricted random permutations. We do this by relating the external coascent of a set of block-preserving permutations to that of the set of block permutations that induces it. Recall that $x = \lfloor \log n \rfloor$.

LEMMA 8.5. *For all* $\Xi_0,\dots,\Xi_{n-1}\in\mathrm{Perm}(M)$ *with induced block-preserving permutations* $\xi_0,\dots,\xi_{n-1}\in\mathrm{Perm}(V)$, $\Lambda(\xi_0,\dots,\xi_{n-1})\le x\cdot\Lambda(\Xi_0,\dots,\Xi_{n-1})$.

Proof. Fix $\sigma\in\mathrm{Perm}(V)$ arbitrarily and let $R\subseteq\mathrm{Perm}(M)$ be the multiset obtained as follows: for each tuple $(r_0,\dots,r_{n/x-1})\in B_0\times\cdots\times B_{n/x-1}$, where r_i for $i=0,\dots,n/x-1$ is called a *representative* of its block B_i, add to R (one copy of) the block permutation Ψ that arranges the blocks in the order in which their representatives occur in σ (i.e., for $0\le i,j\le n/x-1, \Psi^{-1}(i)<\Psi^{-1}(j)\iff\sigma^{-1}(r_i)<\sigma^{-1}(r_j)$). We call $r_0,\dots,r_{n/x-1}$ the *defining vertices* of (that copy of) Ψ. Now, for every block permutation $\Xi\in\mathrm{Perm}(M)$ with induced block-preserving permutation ξ,

$$\sum_{\Psi\in R}\lambda(\Xi,\Psi)\ge\frac{|R|}{x}\lambda(\xi,\sigma).$$

To see this, note that each element of a fixed longest common subsequence of $\xi(0),\dots,\xi(n-1)$ and $\sigma(0),\dots,\sigma(n-1)$ contributes 1 to $\lambda(\Xi,\ \Psi)$ if it is a defining vertex of Ψ and that each $v\in V$ is a defining vertex of exactly $|R|/x$ permutations $\Psi\in R$. Summing the inequality above for Ξ equal to $\Xi_0,\dots,\Xi_{n-1}$ produces

$$\begin{aligned}\sum_{v=0}^{n-1}\lambda(\xi_v,\sigma)&\le\frac{x}{|R|}\sum_{v=0}^{n-1}\sum_{\Psi\in R}\lambda(\Xi_v,\Psi)=\frac{x}{|R|}\sum_{\Psi\in R}\sum_{v=0}^{n-1}\lambda(\Xi_v,\Psi)\\&\le\frac{x}{|R|}\sum_{\Psi\in R}\Lambda(\Xi_0,\dots,\Xi_{n-1})=x\cdot\Lambda(\Xi_0,\dots,\Xi_{n-1}).\qquad\square\end{aligned}$$

We can now state the main result of our paper and finally justify its title.

THEOREM 8.2. *A maximum flow in a network with n vertices can be computed deterministically using* $O(n^{8/3}(\log n)^{4/3})$ *flow operations and* $O(n^3/\log n)$ *time.*

Proof. According to Lemma 8.4(b), used with $h=n/x$, n block permutations $\Xi_0,\dots,\Xi_{n-1}\in\mathrm{Perm}(M)$ with $\Lambda(\Xi_0,\dots,\Xi_{n-1})=O(n(n/\mathrm{log}n)^{2/3})$ can be constructed in $O(n^2/\log n)$ time. By Lemmas 8.3 and 8.5, if the algorithm is executed with the adjacency lists ordered according to the block-preserving permutations induced by $\Xi_0,\dots,\Xi_{n-1}$, then $\sharp ptr=O(n^{8/3}(\log n)^{1/3})$. The claim now follows from Lemmas 7.5 and 8.2. $\square$

9. Parallel algorithms. Since our solution to the current-edge problem parallelizes trivially on most parallel machines and since the current-edge problem is the only bottleneck in our algorithms on dense graphs, it is possible to crank out a variety of parallel algorithms for the maximum-flow problem that have optimal speedup, as compared with their sequential counterparts. We give one example in Theorem 9.1 below. Since we parallelize only the current-edge data structure and execute all other parts of the algorithms sequentially as before, optimal speedup can be attained only for a moderately small number of processors, as is to be expected in view of the P-completeness of the maximum-flow problem [GSS82]. Previous work on parallel algorithms for computing maximum flows is described in [SV82], [GT88], [GT89], and [Go91]. No parallel algorithm for the maximum-flow problem with optimal speedup (using more than a constant number of processors) was previously known.

THEOREM 9.1. *For* $p=O(n^{1/3}(\log n)^{-7/3})$, *a maximum flow in a network with n vertices can be computed in (optimal)* $O(n^3/(p\log n))$ *time on a network of* $2p-1$ *processors interconnected to form a complete binary tree.*

Proof. The processor at the root of the tree stores a copy of all variables. In addition, each of the p leaf processors has a copy of the vector z, and a copy of the arrays r' and H' is distributed among the leaf processors, the ith leaf processor, for $i = 1, \ldots, p$, storing the columns of r' and H' numbered $(i-1)$, $(i-1)+p$, $(i-1)+2p$, etc. The root processor essentially carries out the algorithm of Theorem 8.2 sequentially. Each update of r' and H' is broadcast to the leaf processors and recorded by the relevant leaf processor. A call of $ce(v)$ (originating at the root) is also broadcast to the leaf processors and causes each of them to advance its copy of $z[v]$, looking only at the columns stored locally, until it encounters an admissible edge, runs out of edges, or is interrupted. A successful processor sends the admissible edge found up the tree towards the root. Whenever two edges meet in the tree, the one coming from the right is discarded. The root, upon receipt of the surviving edge, which is $ce(v)$, broadcasts it to the leaves. This signal interrupts the leaf processors and allows them to reset $z[v]$ to the correct value.

This implementation allows a sequence of q operations on the current-edge data structure to be processed in $O(q \log p + n^3/(p \log n))$ time. The algorithm of Theorem 8.2 uses $q = O(n^{8/3}(\log n)^{1/3})$ operations, giving a total time of $O(n^{8/3}(\log n)^{4/3} + n^3/(p \log n))$. For $p = O(n^{1/3}(\log n)^{-7/3})$, this is $O(n^3/(p \log n))$. □

10. Open problems. (1) Does the current-edge problem have an $o(nm)$-time solution for $m = o(n^2/\log n)$? A positive answer to this question would extend the range of $o(nm)$ algorithms below $m = \Omega(n^2/\log n)$.

(2) Can the $O(n^2 \log n/\log(2 + n(\log n)^2/m))$ term be dropped in the analysis of the number of PTR events? A positive answer to this question would extend the range of $O(nm)$ algorithms below $m = \Omega(n(\log n)^2)$.

(3) Is there an $o(nm \log n)$ maximum-flow algorithm for $m = o(n \log n/\log\log n)$?

REFERENCES

[AHU74] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison–Wesley, Reading, MA, 1974.

[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.

[Al90] 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 Proc. 30th Annual Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1989, pp. 118–123.

[CH95] ———, *A randomized maximum-flow algorithm*, SIAM J. Comput., 24 (1995), pp. 203–226.

[CLR90] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, MIT Press, Cambridge, MA, and McGraw–Hill, New York, 1990.

[FF62] L. R. Ford, Jr., and D. R. Fulkerson, *Flows in Networks*, Princeton University Press, Princeton, NJ, 1962.

[Go91] A. V. Goldberg, *Processor-efficient implementation of a maximum flow algorithm*, Inform. Process. Lett., 38 (1991), pp. 179–185.

[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] ———, *A parallel algorithm for finding a blocking flow in an acyclic network*, Inform. Process. Lett., 31 (1989), pp. 265–271.

[GSS82] L. M. Goldschlager, R. A. Shaw, and J. Staples, *The maximum flow problem is log space complete for P*, Theoret. Comput. Sci., 21 (1982), pp. 105–111.

[GLS88] M. Grötschel, L. Lovász, and A. Schrijver, *Geometric Algorithms and Combinatorial Optimization*, Springer-Verlag, Berlin, 1988.

[HR90] T. Hagerup and C. Rüb, *A guided tour of Chernoff bounds*, Inform. Process. Lett., 33 (1990), pp. 305–308.

[Ka74] A. V. Karzanov, *Determining the maximal flow in a network by the method of preflows*, Soviet Math. Dokl., 15 (1974), pp. 434–437.

[KRT94] V. King, S. Rao, and R. Tarjan, *A faster deterministic maximum flow algorithm*, J. Algorithms, 17 (1994), pp. 447–474.

[Se77] R. Sedgewick, *Permutation generation methods*, Comput. Surveys, 9 (1977), pp. 137–164.

[SV82] Y. Shiloach and U. Vishkin, *An $O(n^2 \log n)$ parallel MAX-FLOW algorithm*, J. Algorithms, 3 (1982), pp. 128–146.

[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.

[Ta89] R. E. Tarjan, personal communication, September 1989.

SIAM J. COMPUT.
Vol. 25, No. 6, pp. 1171–1195, December 1996

003

A DETERMINISTIC POLY(LOGLOGN)-TIME N-PROCESSOR ALGORITHM FOR LINEAR PROGRAMMING IN FIXED DIMENSION*

MIKLOS AJTAI[†] AND NIMROD MEGIDDO[‡]

Abstract. It is shown that for any fixed number of variables, linear-programming problems with n linear inequalities can be solved deterministically by n parallel processors in sublogarithmic time. The parallel time bound (counting only the arithmetic operations) is $O((\log\log n)^d)$, where d is the number of variables. In the one-dimensional case, this bound is optimal. If we take into account the operations needed for processor allocation, the time bound is $O((\log\log n)^{d+c})$, where c is an absolute constant.

Key words. parallel computation, expander graph, parallel random-access machine (PRAM), linear programming

AMS subject classifications. 68U05, 90C05

1. Introduction.

The general linear-programming problem is known to be P-complete [6], so it is interesting to investigate the parallel complexity of special cases. One important case is when the number of variables (the dimension) d is fixed while the number of inequalities n grows. Megiddo [11] showed that this problem can be solved in $O(n)$ time for any fixed d. Clarkson [4] and Dyer [8] improved the dependence of the constant on d. The general search technique proposed in [11] provides poly-logarithmic algorithms with n processors for any fixed d (see [12]). Deng [5] gave an $O(\log n)$ algorithm with $n/\log n$ processors for the case where $d = 2$. It was not previously known whether the problem could be solved in $o(\log n)$ time with n processors for any $d > 1$. However, Alon and Megiddo [3] showed that on a probabilistic concurrent-read/concurrect-write parallel random-access machine (CRCW PRAM) with n processors, the problem can be solved by a Las Vegas algorithm almost surely in constant time. In this paper, we show for the first time that for any d, the problem can be deterministically solved in $O((\log\log n)^d)$ time on n processors, if we count only the arithmetic operations. If we take into account the steps necessary for processor allocation, then our time bound is $O((\log\log n)^{d+c})$, where c is an absolute constant. We describe our model of computation in §3. We note that the simple case of $d = 1$ is equivalent to the problem of finding the maximum of n elements, which requires $\Omega(\log\log n)$ time on n processors.

2. Preliminaries.

We first review some known facts about expander graphs. Let $G = (V, E)$ be any graph. For any nonnegative integer r, denote by $G^r = (V, E^r)$ a graph where $(u, v) \in E^r$ if and only if there exists in G a path of length less than or equal to r from u to v. For any $S \subset V$, the r-neighborhood of S, $N_r(S) = N_r(S; G)$, is defined to be the set of all vertices v such that either $v \in S$ or there exists a $u \in S$ with $(u, v) \in E^r$.

DEFINITION 2.1. *A graph $G = (V, E)$ is called* an expander with expansion coefficient α *if for every $S \subset V$ such that $|S| \le \frac{1}{2}|V|$, we have $|N_1(S)| \ge \alpha|S|$.*

* Received by the editors July 27, 1992; accepted for publication (in revised form) February 13, 1995. This research was supported in part by ONR contracts N00014-91-C-0026 and N00014-94-C-0007.

[†] IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120-6099.

[‡] IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120-6099 and School of Mathematical Sciences, Tel Aviv University, Tel Aviv, Israel.

The work of Gabber and Galil [9] provides for every $n = m^2$, $m = 1, 2, \ldots$, an explicit construction of a 6-regular graph which is a 1.03-expander. For any sufficiently large n we can get a 1.02-expander graph on n vertices with maximum degree less than 7 by first taking a 6-regular 1.03-expander on n' vertices, where n' is the smallest square greater than n, and then discarding $n' - n$ arbitrary vertices from it. Let $\epsilon = 0.02$

PROPOSITION 2.2. *For every sufficiently large positive integer n and every positive integer r, there exists a graph $G = (V, E)$ on n vertices with maximum degree less than 7^r such that for every $S \subset V$,*

$$|N_1(S; G)| > \min\{(1+\epsilon)^r |S|, n/2\}.$$

Proof. Let G_0 be a graph on n vertices with a maximum degree of 6 with expansion coefficient $\alpha > 1.1$. As we have already noted, an explicit construction for such a graph is given in [9]. In §6, we describe how the construction can be carried out on a CRCW PRAM with n processors in a constant number of steps, where each processor can perform arithmetic operations on numbers not greater than n. Let $G = (G_0)^r$. The maximum degree d_r in G_r is not greater than $\sum_{i=1}^{r} 6^i < 7^r$. Moreover, for every j, if $|N_j(S; G)| \le n/2$, then $|N_{j+1}(S; G)| > (1+\epsilon)|N_j(S; G)|$, and the proof follows by induction. □

COROLLARY 2.3. *In the graph $(G_0)^r$, if A and B are sets of vertices with cardinalities greater than $n/(1+\epsilon)^{r/2}$, then there exists an edge between them.*

Proof. It follows from Corollary 2.2 that

$$|N_{r/2}(A; G_0)| > \min\{(1+\epsilon)^{r/2}|A|, n/2\} = \min\{n, n/2\} = n/2$$

and, similarly, $N_{r/2}(B; G_0) > n/2$. This implies that $N_{r/2}(A; G_0) \cap N_{r/2}(B; G_0) \ne \emptyset$, and hence in G_0, there is a path of length less than or equal to r between A and B or, equivalently, an edge of $(G_0)^r$. □

Let $c = 2\log_{1+\epsilon} 7$ so that $(1+\epsilon)^{c/2} = 7$.

PROPOSITION 2.4. *For every sufficiently large n and every positive integer r, there exists a graph G of degree less than $d = 7^r$ with the following property: if $t^c = d$, then every two disjoint sets A and B of vertices of G such that $|A| = |B| = n/t$ are connected by an edge.*

Proof. The proof follows directly from Corollary 2.3 since $t = d^{1/c} = 7^{r/c} = (1+\epsilon)^{r/2}$. □

COROLLARY 2.5. *In the expander of Proposition 2.4, for every two disjoint sets of vertices A and B of vertices such that $|A| = |B| = 2n/t$, there exist more than n/t edges between A and B.*

Proof. Suppose to the contrary that the number of edges between A and B is less than or equal to n/t. Let A_1 be the subset of A that consists of those vertices that are not adjacent to any vertex in B. Since $|A_1| \ge n/t$, by Proposition 2.4, there is an edge between A_1 and B—a contradiction. □

3. Linear programming in the plane.

Consider the linear-programming problem with two variables in the form

$$(P^2) \qquad \begin{array}{rll} \text{Minimize} & y & \\ \text{subject to} & y \ge a_i x + b_i & (i \in N_+), \\ & y \le a_i x + b_i & (i \in N_-), \\ & \ell \le x \le h, & \end{array}$$

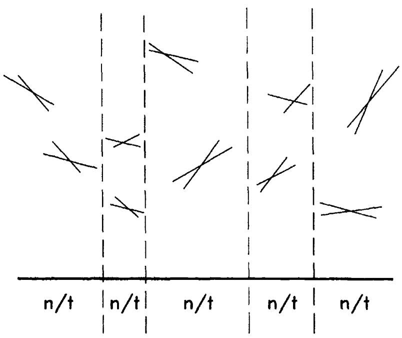


FIG. 1.

where $|N_+| + |N_-| = n$ and $\{\ell, h\} \subset [-\infty, \infty]$. Any two-variable linear-programming problem can be reduced to this form in $O(\log\log n)$ time with n processors. The algorithm proposed by Dyer [7] and Megiddo [10] provides a method of discarding $\frac{1}{4}$ of the set of constraints with an effort of computing one median and two maxima in sets of at most n elements. It was shown by Ajtai, Komlós, Steiger, and Szemerédi [2] that selection can be done in $O(\log\log n)$ time in Valiant's parallel-comparison-tree model with n processors. The selection steps of our algorithm are implemented in this model. All of the other steps can be implemented on a CRCW PRAM. Deng [5] gave a parallel algorithm which runs in $O(\log n)$ time using $O(n/\log n)$ processors. In fact, his algorithm applies the procedure of [7, 10] until the number of remaining constraints allows for computation of the entire convex hull in $O(\log n)$ time with $O(n/\log n)$ processors. Such an approach cannot yield an $o(\log n)$-time bound. Our approach discards *increasing* proportions of the set of remaining constraints without resorting to the computation of the entire convex hull of the remaining set at a relatively early stage.

Suppose we are left with the lines

$$L_i = \{(x, y) \mid y = a_i x + b_i\} \quad (i \in N'_+ \cup N'_-),$$

where $N'_+ \subset N_+$ and $N'_- \subset N_-$. Let $n = |N_+| + |N_-|$ denote the revised number of constraints, and we continue to employ p processors (where p is the initial number of constraints). We now describe how a large number of constraints can be further discarded. Denote $q = p/n$ (where p is the initial number of processors). The treatment of the two classes of constraints is very similar, so we describe only the case of N'_+.

Let d be the largest power of 7 that is smaller than $q^{c/(c+1)}$. Let t be defined by $t^c = d$. Consider an expander graph $G = (V, E)$ of degree smaller than d, with vertices corresponding to the lines in N'_+, and that has the properties asserted in Proposition 2.4 and Corollary 2.5.

For every edge $(i, j) \in E$ $(i, j \in N'_+)$, if L_i and L_j are not parallel, consider the intersection coordinate $\xi_{ij} = -(b_i - b_j)/(a_i - a_j)$. Denote by C the set of these intersection points. Obviously, $|C| \leq \frac{1}{2}nd < p$, so all the points in C can be computed in constant time with our p processors. Partition C into intervals by points $-\infty = a_0 < x_1 < x_2 < \cdots < x_s = \infty$ so that each interval $[x_{i-1}, x_i]$ $(i = 1, \ldots, s)$ contains less than n/t points. (See Figure 1.) This can be achieved with $s < nd/(n/t) = t^{c+1} = q$.

PROPOSITION 3.1. *If A and B are disjoint sets of lines, each containing at least $2n/t$ elements, then there does not exist a k $(1 \leq k \leq s)$ such that $\xi_{ij} \in [x_{k-1}, x_k]$ for* *all $L_i \in A$ and $L_j \in B$.*

Proof. The proof follows from the fact that the number of such intersection points that are also in C is at least n/t by Corollary 2.5, while each interval contains less than n/t points of C. □

Given the set C, we wish to determine in which of the intervals $[x_{k-1}, x_k]$ an optimal solution x^* might lie. Denote

$$f_+(x) = \max\{a_i x + b_i \mid i \in N'_+\},$$
$$f_-(x) = \min\{a_i x + b_i \mid i \in N'_-\}.$$

Note that an optimal solution must satisfy the following: $x^* \in [\ell, h]$, $f_+(x^*) \le f_-(x^*)$, and $f_+(x^*)$ is minimal. Since $f_+(x)$ and $f_+(x) - f_-(x)$ are convex, we can test any value of x with at most three computations of a maximum in a set of cardinality n (see [10]) and conclude with one of the following possibilities: (i) the problem has no feasible solution; (ii) x is an optimal solution; (iii) if x^* is an optimal solution, then $x^* > x$; (iv) if x^* is an optimal solution, then $x^* < x$. Since $p/s > n$, we have at least n processors per point x_k, so in $O(\log\log n)$ time, we can locate the optimum in one of our intervals.

Suppose we have identified an interval $[u, v]$ where the minimum is attained, and there are less than n/t intersection points of C over $[u, v]$. Consider the orders induced on the set of lines by their intersections with the lines $\{x = u\}$ and $\{x = v\}$. Call them the u-order and the v-order, respectively. For every k $(k = 1, \ldots, n)$, denote by U_k the set of the k lowest lines in the u-order and denote by V_k the set of the k lowest lines in the v-order.

PROPOSITION 3.2. *For every k $(k = 1, \ldots, n)$, the symmetric difference*

$$U_k \ominus V_k = (U_k \setminus V_k) \cup (V_k \setminus U_k)$$

contains at most $4n/t$ lines.

Proof. Since $|U_k \setminus V_k| = |V_k \setminus U_k|$, it follows that if, to the contrary, $|U_k \ominus V_k| > 4n/t$, then

$$|U_k \setminus V_k| = |V_k \setminus U_k| > 2n/t\ .$$

Note that all the intersections ξ_{ij} of a line $L_i \in U_k \setminus V_k$ with a line $L_j \in V_k \setminus U_k$ must be in $[u, v]$—a contradiction to Proposition 3.1. □

Let $\bar{U_k}$ and $\bar{V_k}$ be the complements of U_k and V_k, respectively, in the set of all n lines. (See Figure 2.)

PROPOSITION 3.3. *For $k = n - 4n/t$, there exists at least one line in $\bar{U_k} \cap \bar{V_k}$.*

Proof. The claim is trivial if $U_k = V_k$; otherwise, since $U_k \ominus V_k = \bar{U_k} \ominus \bar{V_k}$, by Proposition 3.2,

$$|\bar{U_k} \cap \bar{V_k}| = |\bar{U_k} \cup \bar{V_k}| - |U_k \ominus V_k| \ge (4n/t + 1) - 4n/t = 1. \qquad \square$$

PROPOSITION 3.4. *Let $k = n - 4n/t$. If L is any line in $\bar{U_k} \cap \bar{V_k}$, then over the interval $[u, v]$, the line L lies above at least $n - 8n/t$ lines.*

Proof. The line L lies above every other line, except possibly for some lines in $\bar{U_k} \cup \bar{V_k}$, but $|\bar{U_k} \cup \bar{V_k}| \le 8n/t$. □

COROLLARY 3.5. *The number of lines can be reduced from n to no more than $8n/t$ in $O(\log\log n)$ time.*

Proof. A line $L \in \bar{U_k} \cap \bar{V_k}$ $(k = n - 4n/t)$ can be found as follows. Compute the set $\bar{U_k}$ by selecting the kth smallest element relative to the u-order and then find the

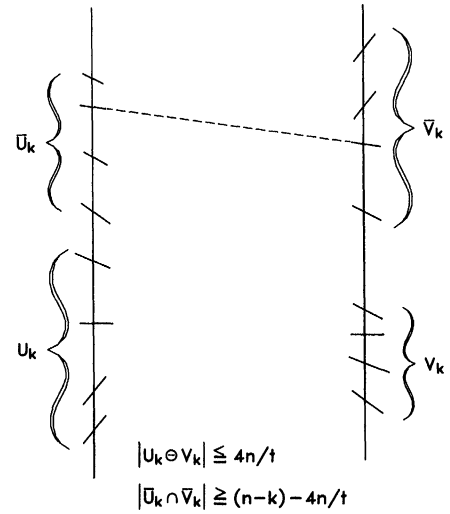

$\bar{U}_k$
$\bar{V}_k$
U_k
V_k
$|U_k \ominus V_k| \leqq 4n/t$
$|\bar{U}_k \cap \bar{V}_k| \geqq (n-k)-4n/t$

FIG. 2.

line L which is the maximum in $\bar{U_k}$ relative to the v-order. Given L, we can compare it with all of the other lines and discard those lines which are smaller in both orders. The number of remaining lines will be at most $8n/t$. □

THEOREM 3.6. *The linear-programming problem with two variables can be solved in $O((\log\log n)^2)$ time.*

Proof. The scheme we have described so far reduces the number of constraints from n to $8n/t(n)$, so it works only when $8n/t(n) < n$. In order to satisfy the latter condition, we start the algorithm by running a constant number of iterations of the algorithm of [7, 10], where $\frac{1}{4}$ of the set of constraints is discarded in each iteration. This constant number is determined from $t(n) = (p/n)^{1/(c+1)} > 8$, i.e., $n/p < 8^{-c-1}$, and the number is $\log_{3/4} 8^{-c-1}$.

Recall that $t = t(n) = (p/n)^{1/(c+1)}$, so the value of n is reduced in one iteration to

$$n' = 8n^{1+1/(1+c)}p^{-1/(c+1)},$$

so the next value of t is

$$t' = \left(\frac{1}{8}p^{1+1/(c+1)}n^{-1-1/(1+c)}\right)^{1/(c+1)} = 8^{-1/(c+1)}t^{1+1/(1+c)}.$$

Thus, after k iterations, the value of t is

$$t^{(k)} = \beta^{1+\gamma+\cdots+\gamma^{k-1}}t^{\gamma^k},$$

where $\beta = 8^{-1/(c+1)}$ and $\gamma = 1 + 1/(c+1)$, and this implies that the number of iterations is $O(\log\log n)$. □

4. The three-dimensional case. The linear-programming problem with three variables is formulated as follows:

$$(P^3) \qquad \begin{array}{rlr} \text{Minimize} & z & \\ \text{subject to} & z \geq a_i x + b_i y + c_i & (i \in N_+), \\ & z \leq a_i x + b_i y + c_i & (i \in N_-), \\ & 0 \leq a_i x + b_i y + c_i & (i \in N_0), \end{array}$$

where $|N_+| + |N_-| + |N_0| = n$.

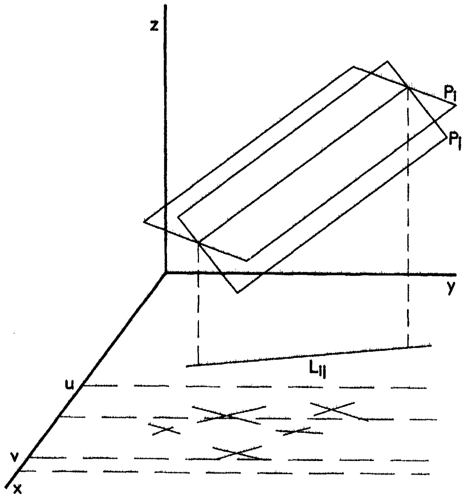


FIG. 3.

As in the two-dimensional case, we proceed by discarding increasing proportions of the set of constraints. The three sets N_+, N_-, and N_0 can be handled independently. We describe only the processing of N_+. Suppose we are currently left with n planes

$$P_i = \{(x, y, z) \mid z = a_i x + b_i y + c_i\} \quad (i \in N'_+ \subset N_+)$$

and there are p processors. For every pair of planes (P_i, P_j), if the planes are not parallel, let

$$L_{ij} = \{(x, y) \mid a_i x + b_i y + c_i = a_j x + b_j y + c_j\},$$

i.e., L_{ij} is the projection of the line of intersection of P_i and P_j into the (x, y)-plane. (See Figure 3.)

We now use an expander graph $G = (N'_+, E)$ of maximum degree less than t_1^c (the dependence of t_1 on n and p will be explained later) whose vertices represent the planes and with the property that if A and B are disjoint subsets of N'_+, each of cardinality of at least $2n/t_1$, then the number of edges between A and B is at least n/t_1. Let

$$D = \{L_{ij} \mid (i, j) \in E\}.$$

Note that the number of pairs (i, j) such that $L_{ij} \in D$ is less than nt_1^c.

We now use an expander graph $G' = (D, E')$ (i.e., the vertices correspond to the lines in D) with maximum degree less than t_2^c (the dependence of t_2 on n and p

will be explained later) so that between any two sets, each of cardinality of at least $2|D|/t_2$, the number of edges is at least $|D|/t_2$. In view of Corollary 2.5, this is true if $t_2 \le (p/(nt_1^c))^{1/(c+1)}$ since $|D| \le nt_1^c$.

Denote by C the set of intersection points of pairs of lines corresponding to the edges of G'. We proceed as in the two-dimensional case (as if D were the total set of lines). We partition the x-axis into t_2^{c+1} intervals in the same way and find one interval $[u, v]$ of the partition such that, without loss of generality, the given instance of (P^3) may be restricted to the stripe[1] $\{(x, y) \mid u \le x \le v\}$. The testing algorithm that we use to select the required stripe is described in detail in [11, §4, pp. 123–126]. We can decide with this testing algorithm where the solution of (P^d) is relative to a given hyperplane, that is, whether there is a solution on the hyperplane and, if not, then which of the two half-spaces determined by the hyperplane contains the solution. The algorithm uses only the solution of three instances of the $(d-1)$-dimensional problem if the hyperplane is in the $(d-1)$-dimensional space. (We will use this algorithm for the solution of the d-dimensional problem as well). We may now conclude that the required stripe can be selected by solving at most three instances of the two-dimensional problem. We also know that the two orders on D (induced by the intersection points with $\{(x, y) \mid x = u\}$ and $\{(x, y) \mid x = v\}$) are almost the same in the sense that for every k ($k = 1, \dots, |D|$),

$$|U_k \ominus V_k| \le 4|D|/t_2$$

(see Proposition 3.2).

Consider the following two partitions of D into $r < t_2/8$ intervals of length $\delta > 8|D|/t_2$: (i) intervals $I_1, \dots, I_r$ relative to the u-order and (ii) intervals $J_1, \dots, J_r$ relative to the v-order.

PROPOSITION 4.1. *For every k ($k = 1, \dots, r$), $I_k \cap J_k \ne \emptyset$.*

Proof. Since $|U_k \ominus V_k| \le 4|D|/t_2$ ($k = 1, \dots, |D|$),

$$I_k = U_{k\delta} \setminus U_{(k-1)\delta}, \quad \text{and} \quad J_k = V_{k\delta} \setminus V_{(k-1)\delta},$$

it follows that

$$|I_k \ominus J_k| \le |(U_{k\delta} \ominus V_{k\delta}) \cup (U_{(k-1)\delta} \ominus V_{(k-1)\delta})| \le 2(4|D|/t_2) < \delta,$$

so

$$|I_k \cap J_k| \ge |I_k \cup J_k| - |I_k \ominus J_k| > 0. \qquad \square$$

It follows that for each k ($k = 1, \dots, r$), there exists a line $\ell_k \in I_k \cap J_k$ so that the members of $M = \{\ell_1, \dots, \ell_r\}$ do not intersect in the stripe $\{(x, y) \mid u \le x \le v\}$. These lines partition the stripe into "trapezoids." (See Figure 4.)

Remark 4.2. A suitable set M can be constructed on a CRCW PRAM in $O(\log \log n)$ time. First, the intervals are constructed by solving r selection problems. Suppose each member of D knows the intervals that it belongs to relative to the two orders. Each member of $I_k \cap J_k$ now attempts to write its name in a cell representing ℓ_k, and one succeeds.

PROPOSITION 4.3. *There are at most 5δ pairs (i, j) (not necessarily in E) such that the trapezoid bordered by the ℓ_k and ℓ_{k+1} is intersected by L_{ij}.*

[1] Note that we may have $u = -\infty$ or $v = \infty$.

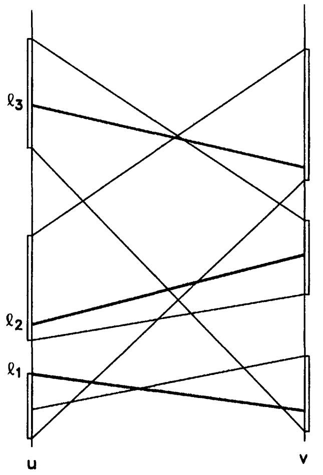


FIG. 4.

Proof. The proof follows from the fact that this trapezoid may intersect only the lines in the set

$$I_k \cup I_{k+1} \cup J_k \cup J_{k+1} \cup (U_{(k+1)\delta} \ominus V_{(k+1)\delta}) \cup (U_{(k-1)\delta} \ominus V_{(k-1)\delta})$$

whose cardinality is at most $4\delta + 2(4|D|/t_2)$. □

Next, we identify one trapezoid to which problem (P^3) may be restricted without loss of generality. Furthermore, we can divide this trapezoid into two triangles and restrict our attention to one of them, which we denote by T.

PROPOSITION 4.4. *If $5\delta < n/t_1$, then for every pair (A, B) of disjoint sets of planes such that $|A|, |B| > 2n/t_1$, there is at least one line L_{ij} such that $P_i \in A$ and $P_j \in B$, and L_{ij} does not intersect T.*

Proof. We know that for such an A and B, there are at least n/t_1 lines L_{ij} in D such that $P_i \in A$ and $P_j \in B$. On the other hand, by Proposition 4.3, T is intersected by at most 5δ lines L_{ij}. □

Thus we need to choose t_1 and t_2 so that $40|D|/t_2 < n/t_1$. Hence we require $40t_1^c/t_2 < 1/t_1$, i.e., $40t_1^{c+1} < t_2$, and since $t_2 < (p/|D|)^{1/(c+1)}$, it suffices that

$$t_1 < \tfrac{1}{40}\left(\frac{p}{n}\right)^{\frac{1}{(c+1)(c+2)}} \quad \text{and} \quad t_2 = \left(\frac{p}{n}\right)^{\frac{1}{(c+2)}}.$$

At each of the three vertices[2] of T, there is a natural linear order on the set of hyperplanes induced by the z-coordinate. We may apply the argument that we used in the two-dimensional case to any pair of orders as we did with the u-order and the v-order. For every k $(k = 1, \ldots, n)$, let U_k, V_k, and W_k denote the sets of k lowest planes relative to these three orders.

PROPOSITION 4.5. *For every k $(k = 1, \ldots, n)$, the set*

$$S_k = (U_k \ominus V_k) \cup (U_k \ominus W_k) \cup (V_k \ominus W_k)$$

contains at most $12n/t_1$ elements.

[2] The case of an "infinite" triangle can be easily handled as well.

Proof. Each member of $U_k \setminus V_k$ intersects each member of $V_k \setminus U_k$ in T, each member of $V_k \setminus W_k$ intersects each member of $W_k \setminus V_k$ in T, and each member of $W_k \setminus U_k$ intersects each member of $U_k \setminus W_k$ in T. Since $|U_k \setminus V_k| = |V_k \setminus U_k|$, $|V_k \setminus W_k| = |W_k \setminus V_k|$, and $|W_k \setminus U_k| = |U_k \setminus W_k|$, it follows from our choice of t_1 that the cardinality of each of these six sets is not greater than $2n/t_1$. □

PROPOSITION 4.6. *For $k = n - 12n/t_1$, $\bar{U}_k \cap \bar{V}_k \cap \bar{W}_k \neq \emptyset$.*

Proof. If $U_k = V_k = W_k$, the claim is trivial; otherwise,

$$|\bar{U}_k \cap \bar{V}_k \cap \bar{W}_k| \geq |\bar{U}_k \cup \bar{V}_k \cup \bar{W}_k| - |S_k| \geq (12n/t_1 + 1) - 12n/t_1 = 1. \qquad \square$$

PROPOSITION 4.7. *If $k = n-12n/t_1$ and $P \in \bar{U}_k \cap \bar{V}_k \cap \bar{W}_k$, then over the triangle T, the plane P lies above at least $n - 36n/t_1$ planes.*

Proof. The plane P lies above every plane in $U_k \cap V_k \cap W_k$, but

$$|U_k \cap V_k \cap W_k| = n - |\bar{U}_k \cup \bar{V}_k \cup \bar{W}_k| \geq n - 3(12n/t_1).$$

(See Figure 5.) □

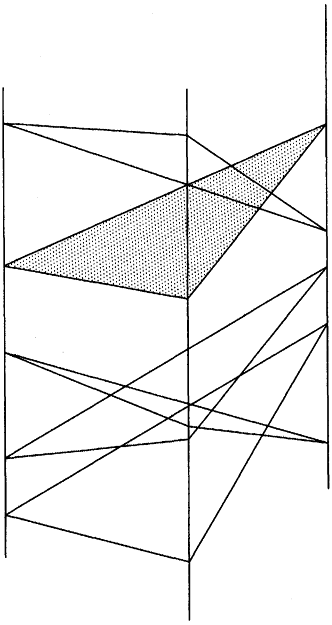


FIG. 5.

COROLLARY 4.8. *The number of planes can be reduced from n to no more than $36n/t_1$ in $O((\log\log n)^2)$ time.*

Proof. The sets $\bar{U}_k$, $\bar{V}_k$, and $\bar{W}_k$ can be computed with a selection algorithm, and then it can be decided separately (and simultaneously) for each member of the union of these sets whether it satisfies the conditions of Proposition 4.7. Since we need to solve linear-programming problems with two variables and we have enough processors for solving all of these problems in parallel, Theorem 3.6 applies, so the effort for one iteration is $O((\log\log n)^2)$. □

THEOREM 4.9. *The linear-programming problem with three variables can be solved in $O((\log\log n)^3)$ time.*

Proof. The value of n is reduced in one iteration to $36n/t_1$, where $t_1 = (p/n)^{1/((c+1)(c+2))}$. As in the proof of Theorem 3.6, the next value of t_1 is

$$t_1' = 36^{-1/((c+1)(c+2))} t_1^{1+1/((c+1)(c+2))},$$

so the number of iterations is $O(\log\log n)$. This implies our claim. □

5. The general *d*-dimensional case. We now consider the general linear-programming problem with d variables, which we formulate as follows.

$$(P^d) \qquad \begin{array}{rll} \text{Minimize} & y & \\ \text{subject to} & y \geq \boldsymbol{a}_i^T \boldsymbol{x} + b_i & (i \in N_+), \\ & y \leq \boldsymbol{a}_i^T \boldsymbol{x} + b_i & (i \in N_-), \\ & 0 \leq \boldsymbol{a}_i^T \boldsymbol{x} + b_i & (i \in N_0), \end{array}$$

where $\boldsymbol{a}_i \in R^{d-1}$ ($i \in N_+ \cup N_- \cup N_0$ and $|N_+| + |N_-| + |N_0| = n$).

5.1. Hyperplane queries. In general, our algorithm works recursively in the dimension. First, as explained in [11], a linear-programming algorithm for problems with $d-1$ variables can be used as an oracle for deciding the position of the set of optimal solutions, if any, relative to any given hyperplane. More precisely, it can be used to solve the following problem.

Problem 5.1. Given an instance of (P^d) and a hyperplane $H = \{\boldsymbol{x} \in R^{d-1} \mid \boldsymbol{a}^T\boldsymbol{x} = b\}$, decide whether (i) the optimal solutions of (P^d), if any, may be assumed to lie in $H_+ = \{\boldsymbol{x} \in R^{d-1} \mid \boldsymbol{a}^T\boldsymbol{x} > b\}$, (ii) the optimal solutions of (P^d), if any, may be assumed to lie in $H_- = \{\boldsymbol{x} \in R^{d-1} \mid \boldsymbol{a}^T\boldsymbol{x} < b\}$, or (iii) a final conclusion can be reached that either H contains an optimal solution, the problem is unbounded on H, or the problem is infeasible. The conclusion in (iii) is reached when the solution of (P^d) with the additional constraint $\boldsymbol{x} \in H$ yields a solution of (P^d) or when the problem is infeasible and the "amount of infeasibility" is minimized on H.

5.2. Locating the solution in a "small" simplex. We now introduce a problem that plays the key role in the algorithm, but we first need to define an oracle for minimizing a function.

DEFINITION 5.2. *Consider a function $f : R^{d-1} \to R \cup \{-\infty\}$. By an* oracle for *$f$, we mean a mechanism that, when presented with a hyperplane H in R^{d-1}, returns information in one of the following forms:* (i) *either the minimum of f lies in H_+ or f is unbounded from below on H_+;* (ii) *either the minimum of f lies in H_- or f is unbounded from below on H_-;* (iii) *either the minimum of f lies in H or f is unbounded from below on H.* (*See Figure* 6.)

Problem 5.3. The following are given: an oracle for a function f as in Definition 5.2, a number p (of processors), and hyperplanes $H_k = \{(\boldsymbol{x}, y) \in R^d \mid y = \boldsymbol{a}_k^T\boldsymbol{x} + b_k\}$ ($k = 1, \ldots, n$). Find either some hyperplane H in R^{d-1} such that the minimum of f lies in H (or f is unbounded on H) or d half-spaces $F_k = \{\boldsymbol{x} \in R^{d-1} \mid \boldsymbol{c}_k^T\boldsymbol{x} + d_k \geq 0\}$ ($k = 1, \ldots, d$) such that

1. either the minimum of f lies in the "simplex"[3] $\Delta = F_1 \cap \cdots \cap F_d$ or f is unbounded from below on Δ, and

2. at most n/t ($t = t(p/n; d)$) pairs (H_i, H_j) of hyperplanes intersect over Δ. (The value of $t(p/n; d)$ will be derived later.)

[3] In general, this intersection may be unbounded.

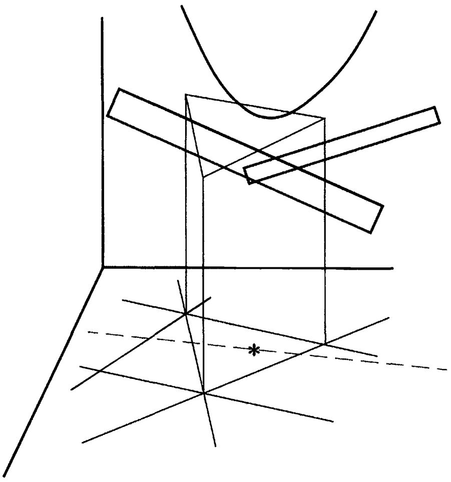


FIG. 6.

Remark 5.4. When $d = 2$, the polyhedron is an interval which may extend to infinity in one direction. In higher dimensions, the polyhedron can be either a simplex or a simplicial cone. In any case, there will be at most d linear orders on the set of H_k's such that if H_i is above H_j in each of these linear orders, then H_i lies above H_j at every point of Δ. Recall that in the case where $d = 2$, we located an interval with $t_1 = (p/n)^{1/(c+1)}$, and in the case where $d = 3$, we located a "triangle" T such that the number of pairs of planes that intersected over T was at most n/t_1, where $t_1 = (p/n)^{1/((c+1)(c+2))}$.

Denote

$$L_{ij} = \{\boldsymbol{x} \in R^{d-1} \mid \boldsymbol{a}_i^T\boldsymbol{x} + b_i = \boldsymbol{a}_j^T\boldsymbol{x} + b_j\}.$$

If H_i and H_j are not parallel, then L_{ij} is a hyperplane in R^{d-1}.

Let $G = (V, E)$ be an expander graph with maximum degree less than τ_1^c (the value of τ_1 will be determined later) whose vertices correspond to the hyperplanes H_k and with the property that every two disjoint subsets $A, B \subset V$ of cardinality n/τ_1 are connected by an edge. Hence if A and B are subsets of V of cardinality at least $2n/\tau_1$, then the number of edges between A and B is at least n/τ_1. Let

$$D = \{L_{ij} \mid (i,j) \in E\}.$$

We have $|D| < n\tau_1^c$. Consider the function $f' : R^{d-2} \to R \cup \{-\infty\}$, defined by

$$f'(x_1, \ldots, x_{d-2}) = \inf_{x_{d-1}} f(x_1, \ldots, x_{d-1}).$$

An oracle for f (in the sense of Definition 5.2) provides an oracle for f' when we extend any hyperplane H in R^{d-2} into a hyperplane in R^{d-1} described by the same equation. Thus, recursively, we can either (i) find a hyperplane $H \subset R^{d-2}$ which contains the minimum of f', and hence its extension into a hyperplane in R^{d-1} contains the minimum of f, or (ii) find $d - 1$ half-spaces $F_k = \{\boldsymbol{x} \in R^{d-2} \mid \boldsymbol{c}_k^T\boldsymbol{x} + d_k \geq 0\}$ $(k = 1, \ldots, d-1)$ with the properties described in Problem 5.3 with respect to the hyperplanes L_{ij} in D:

1. either the minimum of f lies in $\Delta = F_1 \cap \cdots \cap F_{d-1}$ or f is unbounded from below on Δ, and

2. at most $|D|/\tau_2$ ($\tau_2 = t(p/|D|, d-1)$) pairs of L_{ij}'s from D intersect over Δ.

For $k = 1, \ldots, d-1$, let F'_k be the half-space in R^{d-1} parallel to the (x_{d-1})-axis, obtained by extending the half-space F_k into R^{d-1}. Thus the polyhedron Δ is extended into a polyhedral "cylinder" Δ^o in R^{d-1} which contains the minimum of f. Furthermore, the number of pairs of L_{ij}'s intersecting Δ^o is at most $|D|/\tau_2$. Consider the $d-1$ linear orders induced on the set of L_{ij}'s by their x_{d-1} values at the $d-1$ vertices of Δ (or at infinity, as explained above). For $j = 1, \ldots, d-1$ and $k = 1, \ldots, |D|$, denote by U^j_k the set of the k lowest hyperplanes relative to the jth order.

PROPOSITION 5.5. *For any i and j ($1 \le i < j \le d-1$) and for every k ($k = 1, \ldots, |D|$),*

$$|U^i_k \ominus U^j_k| \le 4|D|/\tau_2.$$

Proof. As in Proposition 3.2,

$$U^i_k \ominus U^j_k = (U^i_k \setminus U^j_k) \cup (U^j_k \setminus U^i_k),$$

so if, on the contrary,

$$|U^i_k \ominus U^j_k| > 4|D|/\tau_2,$$

then

$$|U^i_k \setminus U^j_k| = |U^j_k \setminus U^i_k| > 2|D|/\tau_2.$$

Since all the intersections of members of $U^i_k \setminus U^j_k$ with members of $U^j_k \setminus U^i_k$ intersect Δ^o, we reach a contradiction. □

We now consider $d-1$ partitions of D into $r < \tau_2/(4(d-1)(d-2))$ intervals of length $\delta > 4(d-1)(d-2)|D|/\tau_2$: for $i = 1, \ldots, d-1$, a partition into intervals $I^i_1, \ldots, I^i_r$ relative to the ith order.

PROPOSITION 5.6. *For every k ($k = 1, \ldots, r$),*

$$I^1_k \cap I^2_k \cap \cdots \cap I^{d-1}_k \ne \emptyset.$$

Proof. We have

$$|U^i_k \ominus U^j_k| \le 4|D|/\tau_2 \quad \text{and} \quad I^j_k = U^j_{k\delta} \setminus U^j_{(k-1)\delta}.$$

Since

$$I^1_k \cap I^2_k \cap \cdots \cap I^{d-1}_k = (I^1_k \cup I^2_k \cup \cdots \cup I^{d-1}_k) \setminus \bigcup_{i<j} \left(I^i_k \ominus I^j_k\right)$$

and

$$I^i_k \ominus I^j_k \subseteq (U^i_{k\delta} \ominus U^j_{k\delta}) \cup (U^i_{(k-1)\delta} \ominus U^j_{(k-1)\delta}),$$

it follows that

$$|I^1_k \cap I^2_k \cap \cdots \cap I^{d-1}_k| \ge \delta - \sum_{i<j} \left|U^i_{k\delta} \ominus U^j_{k\delta}\right| - \sum_{i<j} \left|U^i_{(k-1)\delta} \ominus U^j_{(k-1)\delta}\right|$$

$$\ge \delta - 2\binom{d-1}{2} \cdot 4|D|/\tau_2 > 0. \qquad \square$$

It follows that for each k ($k = 1, \dots, r$), there exists a hyperplane $L_k^* \in I_k^1 \cap \cdots \cap I_k^{d-1}$ such that the members of $M = \{L_1^*, \dots, L_r^*\}$ do not intersect in the cylinder Δ^o. These hyperplanes partition Δ^o into "prisms."

PROPOSITION 5.7. *There are at most $(2d-1)\delta$ pairs (i,j) such that the prism bordered by L_k^* and L_{k+1}^* is intersected by L_{ij}.*

Proof. The prism may be intersected only by hyperplanes in the set

$$\bigcup_{j=1}^{d-1} \left(I_k^j \cup I_{k+1}^j\right) \cup \bigcup_{i<j} \left(U_{(k+1)\delta}^i \ominus U_{(k+1)\delta}^j\right) \cup \bigcup_{i<j} \left(U_{(k-1)\delta}^i \ominus U_{(k-1)\delta}^j\right),$$

whose cardinality is at most

$$2(d-1)\delta + 2\binom{d-1}{2} \cdot 4|D|/\tau_2 < (2d-1)\delta. \qquad \square$$

We now find one prism that may be assumed to contain the minimum of f. In this process, we might find a hyperplane which contains the minimum. We then divide the prism into $d-1$ "simplices" and restrict our attention to one of them, which we now denote by Δ'.

PROPOSITION 5.8. *If $2d^3|D|/\tau_2 < n/\tau_1$, then for every pair (A,B) of disjoint sets of hyperplanes H_k such that $|A|, |B| > 2n/\tau_1$, there exists at least one L_{ij} such that $H_i \in A$ and $H_j \in B$, and L_{ij} does not intersect Δ'.*

Proof. We choose $\delta > 4(d-1)(d-2)|D|/\tau_2$ such that $(2d-1)\delta < 2n/\tau_1$. For A and B that satisfy our conditions, there are at least n/τ_1 L_{ij}'s in D such that $H_i \in A$ and $H_j \in B$. On the other hand, Δ' is intersected by at most $(2d-1)\delta$ L_{ij}'s. Under the assumption of the proposition, the latter is less than n/τ_1. $\square$

Thus we will choose τ_1 so that $2d^3|D|/\tau_2 < n/\tau_1$. On the other hand, $|D| < n\tau_1^c$, so it suffices that

$$\tau_1 < \left(\frac{\tau_2}{2d^3}\right)^{\frac{1}{c+1}}.$$

We are thus led to the expression for $\tau(d) = t(p/n, d)$ as follows. First, $\tau(1) = q^{1/(c+1)}$ (where $q = p/n$)). Next,

$$\tau(d) = \left(\frac{\tau(d-1)}{2d^3}\right)^{\frac{1}{c+1}}.$$

It follows that

$$\tau(d) = \frac{q^{\epsilon^d}}{2^{\epsilon+\cdots+\epsilon^{d-1}} \prod_{k=1}^d k^{3\epsilon^{d+1-k}}},$$

where $\epsilon = 1/(c+1)$ (not the ϵ of §3).

5.3. Discarding constraints. After we have located the solution of our linear-programming problem in a small simplex, we can discard a large number of constraints as follows. As in the previous sections, we consider members of the three sets N_+, N_-, and N_0 separately. Suppose we are left with a set $N_+' \subseteq N_+$ of n hyperplanes, and we have now found a "simplex" Δ in R^{d-1} which is known to contain the solution and over which at most $(p/n)^{\tau(d)}$ pairs of hyperplanes intersect. We also know d linear orders over N_+' at "vertices" of Δ such that if a hyperplane H lies above another hyperplane H' in all these orders, then H lies above H' over the entire set Δ. For $j = 1, \ldots, d$ and $k = 1, \ldots, n$, denote by U_k^j the set of the k lowest hyperplanes relative to the jth order.

PROPOSITION 5.9. *For every k ($k = 1, \ldots, n$), the set*

$$S_k = \bigcup_{i<j} (U_k^i \ominus U_k^j)$$

contains at most $d(d-1)n/\tau(d)$ elements.

Proof. The proof is similar to that of Proposition 4.5; each of the sets $U_k^i \ominus U_k^j$ contains at most $2n/\tau(d)$ elements. □

PROPOSITION 5.10. *For $k = n - d(d-1)n/\tau(d)$, $\bigcap_{j=1}^d \bar{U}_k^j \neq \emptyset$.*

Proof. The proof is similar to that of Proposition 4.6:

$$\left| \bigcap_{j=1}^d \bar{U}_k^j \right| \geq \left| \bigcup_{j=1}^d \bar{U}_k^j \right| - |S_k| \geq n - k - d(d-1)n/\tau(d). \qquad \square$$

PROPOSITION 5.11. *If $k = n - d(d-1)n/\tau(d)$ and $H \in \bigcap_{j=1}^d \bar{U}_k^j$, then over the simplex Δ, the hyperplane H lies above at least $n - d^2(d-1)n/\tau(d)$ hyperplanes.*

Proof. The hyperplane H lies above every member of $\bigcap_{j=1}^d U_k^j$, but

$$\left| \bigcap_{j=1}^d U_k^j \right| = n - \left| \bigcup_{j=1}^d \bar{U}_k^j \right| \geq n - d^2(d-1)n/\tau(d). \qquad \square$$

COROLLARY 5.12. *The number of planes can be reduced from n to no more than $d^2(d-1)n/\tau(d)$ in one phase, where $(d-1)$-variable linear-programs are solved in parallel, each with a linear number of processors.*

THEOREM 5.13. *For any fixed d, the linear-programming problem with d variables and n inequalities can be solved with n processors in $O((\log\log n)^d)$ time.*

Proof. Denote

$$C = C(d) = \frac{1}{2^{1+1/c} d^3 (d!)^3}.$$

It follows from what we have proven that the value of n can be reduced in one iteration to $n/t(d)$, where

$$t(d) = t(d; p/n) = C(d) \left(\frac{p}{n}\right)^{\epsilon^d}.$$

After one iteration the new value of $t = t(d)$ is $t' = t^{1+\epsilon^d}$, so after k iterations, $t^{(k)} = t^{(1+\epsilon^d)^k}$. □

Remark 5.14. We note that the algorithm has to start with a constant number of iterations that reduce the number of constraints by discarding constant proportions until we get $t > 1$, i.e., if initially $n = p = m$, we need to reduce n until $n < mC^{\epsilon^{-d}}$. In terms of d, the constant proportion is $O(3^{-d^2})$; hence the constant number iterations is $O(-\log C(d) 3^{d^2})$, i.e., $O(3^{d^2} d \log d)$. Then the variable number of iterations is $O(\log\log n/(\log(1+\epsilon))$, i.e., $O((c+1)^d \log\log n)$.

6. The problem of processor allocation. To describe our computational model, first we recall Valiant's comparison-tree model used for measuring the complexity of sorting or selection problems. We formulate it in a way which is suitable for further generalizations. Assume that we have an ordered set with n abstract elements and $n/2$ processors (n is even). At each step, each processor receives two elements, compares them, and reports the result to a central processor. The central processor receives the results and, based on all the information received, decides which comparison should be made and by whom during the next round of comparisons. There is no restriction on the computing ability of the central processor. The sorting/selection problem is solved in k rounds of comparison if after k rounds the central processor knows the answer to the question. For example, if $a_1, \ldots, a_n$ are the elements of the ordered set (not necessarily ordered in this way), then the computation may start by sending a_{2i-1}, a_{2i}, to processor i ($i = 1, \ldots, n$), and after the first round of comparisons, the central processor may decide that processor 1 gets a_7, a_{10}, processor 2 gets a_3, a_9, etc. The motivation of this model is that we want to count only the number of comparisons and not the amount of computation necessary to decide which set of comparisons will be made in the next round.

We may generalize this model for the solution of linear-programming problems. Assume that the coefficients of the constraints in the problem are elements of an abstract ordered field. Each constraint contains a constant number of these abstract elements as coefficients. At the beginning, the coefficients of each constraint are stored at a single processor. Different constraints are stored at different processors. We also assume that at the beginning of each step, each processor holds a constant number of these abstract elements. During a single step, the processor may perform a constant number of arithmetic operations on them, compare the resulting elements, and report the results of the comparisons to the central processor. The central processor, using the reported results of the comparisons, redistributes the elements among the processors and decides which arithmetic operations and comparisons are done in the next step. (The central processor never gets the abstract elements themselves, only the results of the comparisons.) Again, this model measures only the number of arithmetic operations and comparisons necessary for the solution of the linear programming problem and not the amount of computation necessary to decide which arithmetic operations and comparisons must be performed. (Later, we will give a more realistic model which measures this as well.) We get a clearer picture if we separate the arithmetic operations and the comparisons, that is, we assume that in each round either only arithmetic operations or only comparisons are performed. This way, we can measure separately both the number of arithmetic operations and the number of comparisons required for a solution.

Our result in this computational model is that the d-dimensional linear-programming problem can be solved in a way that the number of rounds where we perform only arithmetic operations is $O((\log\log n)^d)$ and the number of rounds where only comparisons are performed is $O((\log\log n)^{d+1})$.

To be able to measure the amount of computation needed to decide which arithmetic operations or comparisons to perform, we will use a CRCW PRAM. We will not, however, describe every step of our computation in this model. We will assume that the selection steps are done in Valiant's comparison tree model. Our algorithm in the CRCW model described below can be performed in $O((\log\log n)^{d+c})$ steps, where c is an absolute constant.

The CRCW PRAM model that we use in this paper consists of processors that communicate with each other according to the following rules. If there are m processors, then the processors are numbered from 1 to m. The number assigned to a processor will be called its *address.* Each processor has a constant number of registers, which may contain positive integers not greater than m. In each step, processors may (simultaneously) read the contents of the first register of any processor. We assume that when processor i reads the contents of the first register of processor j, then the number j is contained in the second register of processor i. Alternatively, the processors may simultaneously try to write into the first register of any processor. (If more than one processor attempts to write in the same register, the one with the smallest address succeeds.) These kinds of steps will be the read/write steps of the processors. Between these steps, the processors can perform a constant number of arithmetic operations on the contents of their own registers.

To handle the real numbers given in the constraints of the linear-programming problem, we assume that each processor also has a constant number of registers, each containing a real number. Here we consider real numbers as elements of an abstract ordered field, so the processors may only perform the arithmetic operation on them and compare them, but the binary bits of the real numbers are not directly available for the processors. We assume that the same rules of reading and writing are valid for these types of registers as for the ones containing integers. Between two read/write steps, each processor is allowed to perform a constant number of operations on the real numbers contained in its registers. We now describe how can we handle certain specific problems in this model.

Expander graphs. The expander graphs that we use were constructed by Gabber and Galil and are described in [9]. If $n = m^2$, then the vertices of the graph are ordered pairs of positive integers (i, j), where $0 \leq i, j < m$. The neighbors of the vertex (i, j) can be computed in a constant number of arithmetic operations modulo m starting from the numbers i and j. We will represent such a graph in the following way. Each vertex will be associated with a processor and the neighbors of that vertex will be listed in the registers of the processor. Therefore, if the number of vertices is not greater than the number of processors, this expander graph can be computed in a constant number of steps on our CRCW PRAM.

Power of a graph. For certain steps of the algorithm, we will need a family of graphs where the maximum degree is not bounded by a constant. In the following, we may assume that graphs may have multiple edges. This makes their representation easier in our model, and it is suitable also for our applications. The graphs will be represented in the following way. If the maximum degree is d, then each vertex v will be associated with a set T_v of processors of size d. (We assume that the addresses of these processors form an interval of length d.) Therefore, each vertex v has an address—the address of the first processor in T_v. The addresses of the neighbors are stored in the processors contained in T_v (the same address may occur several times). If G_1 and G_2 are graphs with the same set of vertices, then their product G_1G_2 is a graph where the edges between x and y are defined in the following way: for each vertex z and each pair of edges e and f such that e connects x and z and f connects z and y, there is a separate edge $E_{z,e,f}$ connecting x and y. It is easy to see that if two graphs G_1 and G_2 are represented this way, then in a constant number of steps, we may compute a representation of their product provided that the number of processors is large enough for the representation of the product. Consequently, if G is a graph, then we may compute the representation of G^k in poly$(\log k)$ steps. We will always have $k \leq \log n$ (where n is the total number of processors), so we will have that G^k

can be always constructed in poly$(\log\log n)$ steps.

Prime numbers. For certain steps of our algorithm, we will need prime numbers. If we have n processors, all of these prime numbers are smaller than $\sqrt{n}$. We may actually compute all of the prime numbers up to $\sqrt{n}$ in a constant number of steps in our model in the following way. We divide the processors into $\sqrt{n}$ intervals, each of length $\sqrt{n}$. The ith interval has to decide whether the number i is a prime or not. Since there are at least i processors in the interval, they can decide this in a constant number of steps: the jth processor checks whether j is a divisor of i. After this, we may assume that if $k \leq \sqrt{n}$, then the kth processor knows whether k is a prime or not, and if necessary, other processors may read this information from its register. If processor j needs a prime from an interval I contained in $[0, \sqrt{n}]$, then each processor whose address is a prime in this interval tries to write its address in the first register of processor i. If there is a prime in I, then the smallest one will appear in the register of j.

All of the steps of the algorithm except the selection procedures can be implemented on a CRCW PRAM. Apart from the specific problems mentioned above (expander graphs, powers of graphs, primes), there is only one step, namely, discarding constraints, whose implementation is not immediate. We solved the linear-programming problem by discarding an increasing proportion of the remaining constraints. In a typical step of the iteration, we assume that the remaining n constraints are stored in an array of n cells $R[1], \ldots, R[n]$, and we discard at least $n(1 - 1/s)$ $(1 \leq s \leq n)$ of them. In order to continue with the algorithm, we need the remaining n/s constraints to be stored in an array whose size is essentially not larger than n/s. More precisely, we need an algorithm for the following problem.

Problem 6.1. Given an array $R[1], \ldots, R[n]$ and a subset $H \subset \{1, 2, \ldots, n\}$ such that $|H| = n/s$, move the contents of each $R[i]$ $(i \in H)$ to some $R[j(i)]$ so that $j(i) \leq n/s^{1-\epsilon}$ and $j(i) \neq j(i')$ for all $i, i' \in H$ $(i \neq i')$.

PROPOSITION 6.2. *For every fixed $\epsilon > 0$, there exists an algorithm for Problem 6.1 which runs in $O(\log\log n)$ time on an n-processor* CRCW PRAM.

Proposition 6.2 implies that wherever we originally reduced the number of constraints from n to n/s, we will be able to reduce it on a CRCW PRAM from n to $n/s^{1-\epsilon}$. This does not affect the upper bounds given in previous sections.

In the following, we will assume that each processor has a register which may contain a single element of a set A. We will say that this element is handled by the processor. We suppose that throughout the computation, each element of A is handled by a single processor (which may be different from step to step) and each processor handles at most one element. We also assume that the processors are ordered in some arbitrary way. The *rank* of a processor is its position relative to the given ordering. According to this definition, if we say that we took the elements of A to the first k processors, then we mean that after executing the algorithm, each element of A will be handled by a processor whose rank is at most k and distinct elements of A will be handled by distinct processors.

DEFINITION 6.3. *For any $\eta > 0$ and any positive integers c, n, and s, let $\Phi_{\eta,c}(n, s)$ denote the following proposition: There exists an algorithm that runs in c time units so that if there are n processors and $|A| \leq n/s$, then after running the algorithm, the number of those elements of A which are not handled by one of the first $n/s^{1-\eta}$ processors is smaller than $n/s^{1+\eta}$.*

PROPOSITION 6.4. *For every sufficiently small $\eta > 0$, there exists a positive integer c such that for all n and s, $\Phi_{\eta,c}(n, s)$ is true.*

The proof will be given later.

PROPOSITION 6.5. *Proposition* 6.4 *implies Proposition* 6.2.

Proof. It suffices to prove Proposition 6.2 for every sufficiently small $\epsilon > 0$ since for smaller ϵ's, the statement of the proposition is stronger. Given a sufficiently small $\epsilon > 0$, let $\eta = \epsilon/2$. Assuming that Proposition 6.4 is true, $\Phi_{\eta,c}(n,s)$ is true. Therefore, there exists an algorithm as explained in Definition 6.3. When we iterate this algorithm, then after each iteration, the number of those elements of A which are not handled by one of the first $n/s^{1-\epsilon}$ processors decreases from n/t to $n/t^{1+\eta}$. After $O(\log\log n)$ iterations, every element of A will be at the place claimed in Proposition 6.2. Note that at the first step, $t = s$. □

Remark 6.6. It suffices to prove Proposition 6.4 for every $s > s_0$, where s_0 is an arbitrary constant. Indeed, it is possible to simulate ns_0 processors with n processors in a constant number of steps, so we may always assume that the number of processors is at least $s_0|A|$. We use the following proposition in the proof of Proposition 6.4.

PROPOSITION 6.7. *For every $\epsilon > 0$, there exist $\epsilon' > 0$ and $c > 0$ such that for all positive integers s, if we have s processors and $|A| \le s^{1-\epsilon}$, then in c steps we can move all of the elements of A to the first $s^{1-\epsilon'}$ processors.*

Our goal is to show that Proposition 6.7 implies Proposition 6.4.

Let $\epsilon > 0$ be any small constant. (Later, we will give an upper bound on ϵ.) We now assume that $|A| = n/s$, and the elements of A are stored at processors with rank not greater than n. For the sake of simplicity, we assume that s is an integer and n is divisible by s, but the proof remains valid in general with minor modifications. We partition the set of processors into n/s intervals of length s. Let A' be the set of all elements of A which occur in an interval where the number of elements from A is less then $s^{1-\epsilon}$. It follows from Proposition 6.7 that there exist $\epsilon' > 0$ and $c > 0$ such that for each interval of this type, we can move the elements of the interval to the first $s^{1-\epsilon'}$ processors of this interval in time c. In this way, all of the elements of A' can be taken to processors with ranks smaller than $(n/s)s^{1-\epsilon'} = ns^{-\epsilon'}$.

Remark 6.8. We note that in order to perform the described step (with regard to moving the elements of A'), we do not have to count the number of elements in the intervals. We simply attempt to apply the algorithm of Proposition 6.7, and we succeed in the intervals that contain the elements of A'. Let X denote the set of the remaining intervals and let A'' denote the set of those elements of A which are at processors belonging to an interval from X (after we have performed the steps based on Proposition 6.7). The set A'' may have essentially the same size as A.

For the next step of the algorithm, we need the following proposition, which is a consequence of the existence of explicitly constructible expander graphs.

PROPOSITION 6.9. *There exist a positive integer r, a $\delta, \gamma \in (0,1)$, and an algorithm that constructs for any positive integers k and m a symmetric nonnegative $m \times m$ matrix $\boldsymbol{B}$ of integer entries such that the following hold:*

1. *The largest eigenvalue of $r^{-k}\boldsymbol{B}$ is* 1, *and the only eigenvector with this eigenvalue is $(1/\sqrt{m})\boldsymbol{e}$ (where $\boldsymbol{e} = (1,\ldots,1)^T$).*
2. *All the other eigenvalues of $r^{-k}\boldsymbol{B}$ lie in the interval $[0,\gamma^k]$.*
3. *If k is sufficiently large relative to r and γ and if $\boldsymbol{v} = (v_1,\ldots,v_m)^T$ is a $(0,1)$-vector such that $\boldsymbol{e}\cdot\boldsymbol{v} \le m/r^{2k}$, then $\|r^{-k}\boldsymbol{B}\boldsymbol{v}\|_2 \le r^{-\delta k}\|\boldsymbol{v}\|_2$.*

Proof. The results of Gabber and Galil [9] imply that there is an integer $d > 1$ and that there exists an explicit construction (for every m) of an $m \times m$ symmetric matrix $\boldsymbol{D}$ with nonnegative-integer entries such that

1. d is an eigenvalue of $\boldsymbol{D}$, and the only eigenvector with eigenvalue d is $(1/\sqrt{m})\boldsymbol{e}$;

2. all the eigenvalues of $\boldsymbol{D}$ lie between $-d$ and d.

Let $\boldsymbol{F} = \boldsymbol{D} + d\boldsymbol{I}$, where $\boldsymbol{I}$ is the identity matrix. We get the eigenvalues of $\boldsymbol{F}$ by adding d to the eigenvalues of $\boldsymbol{D}$. If $r = 2d$ and $\gamma/2d$ is the second-largest eigenvalue of $\boldsymbol{F}$, then it is easy to see that matrix $\boldsymbol{B} = \boldsymbol{F}^k$, as stated in parts 1 and 2 of the proposition. We will show that part 3 is a consequence of the above.

Let $\boldsymbol{v}$ be a (0, 1)-vector such that $\boldsymbol{e}\cdot\boldsymbol{v} \le m/r^k$. Denote by W the linear subspace of all the vectors orthogonal to $\boldsymbol{e}$. Represent $\boldsymbol{v}$ as

$$\boldsymbol{v} = \mu\boldsymbol{e} + \boldsymbol{w},$$

where $\boldsymbol{w} \in W$. Let $\boldsymbol{v}^2, \dots, \boldsymbol{v}^m$ be orthogonal eigenvectors of $r^{-k}\boldsymbol{B}$, all orthogonal to $\boldsymbol{e}$, with eigenvalues $\lambda_2, \dots, \lambda_m$, respectively ($0 \le \lambda_i \le \gamma^k$ for $i = 2, \dots, m$), and represent $\boldsymbol{w} = \alpha_2\boldsymbol{v}^2 + \cdots + \alpha_m\boldsymbol{v}^m$. Obviously,

$$\|r^{-k}\boldsymbol{B}\boldsymbol{v}\| \le \|r^{-k}\boldsymbol{B}\mu\boldsymbol{e}\| + \|r^{-k}\boldsymbol{B}\boldsymbol{w}\|.$$

Now

$$\begin{aligned}\|r^{-k}\boldsymbol{B}\boldsymbol{w}\| &= \left\|\sum_{i=2}^{m}\alpha_i r^{-k}\boldsymbol{B}\boldsymbol{v}^i\right\| = \left\|\sum_{i=2}^{m}\alpha_i\lambda_i\boldsymbol{v}^i\right\| \\ &\le \gamma^k\left\|\sum_{i=2}^{m}\alpha_i\boldsymbol{v}^i\right\| = \gamma^k\|\boldsymbol{w}\| \le \gamma^k\|\boldsymbol{v}\|.\end{aligned}$$

Since

$$\|r^{-k}\boldsymbol{B}\mu\boldsymbol{e}\| = \|\mu\boldsymbol{e}\| = |\mu|\sqrt{m},$$

it follows that

$$\|r^{-k}\boldsymbol{B}\boldsymbol{v}\| \le \mu\sqrt{m} + \gamma^k\|\boldsymbol{v}\|.$$

Now $\boldsymbol{v}$ is a (0, 1) vector, so $\|\boldsymbol{v}\|^2 = \boldsymbol{e}\cdot\boldsymbol{v}$. On the other hand, $\boldsymbol{e}\cdot\boldsymbol{w} = 0$, so

$$\|\boldsymbol{v}\|^2 = \boldsymbol{e}\cdot(\mu\boldsymbol{e}) = \mu m,$$

and we have

$$\mu\sqrt{m} \le m^{-1/2}\|\boldsymbol{v}\|^2.$$

Consequently,

$$\|r^{-k}\boldsymbol{B}\boldsymbol{v}\| \le m^{-1/2}\|\boldsymbol{v}\|^2 + \gamma^k\|\boldsymbol{v}\| \le (m^{-1/2}\|\boldsymbol{v}\| + \gamma^k)\|\boldsymbol{v}\|.$$

Since $\boldsymbol{v}$ is a (0, 1)-vector and $\boldsymbol{e}\cdot\boldsymbol{v} \le m/r^{2k}$, we have $\|\boldsymbol{v}\| \le (mr^{-2k})^{1/2}$. This implies that

$$\|r^{-k}\boldsymbol{B}\boldsymbol{v}\| \le (m^{-1/2}(mr^{-2k})^{1/2} + \gamma^k)\|\boldsymbol{v}\| = (r^{-k} + \gamma^k)\|\boldsymbol{v}\| \ \le r^{-\delta k}\|\boldsymbol{v}\|$$

if k is sufficiently large, where $\delta > 0$ depends only on γ and r. □

In the intervals of X, there are at least $s^{1-\epsilon}$ elements from A''. We apply Proposition 6.9 with $m = n/s = |X|$. We pick k so that $s^{1/8} < r^k < s^{1/4}$; this is possible since $s > s_0$. We take a graph G on the vertex set X whose matrix is $\boldsymbol{B}$. This graph may contain both loops and multiple edges. We try to move the elements of A'' along the edges. More precisely, an element of A'' which is at a processor in an interval I will be moved to a processor of an interval which is connected to I by an edge of the graph. Distinct elements from the same interval I may move to distinct intervals.

Let h be the number of intervals which contain more than $s^{1-\epsilon}$ elements from A''. Later, we will prove the following.

PROPOSITION 6.10. *It is possible to move the elements of A'' in a constant number of steps so that the new arrangement has the following property: if K is the set of intervals which contain more than $s^{1-\epsilon}$ elements from A'', then $|K| < s^{-\delta/5}h$, where δ is the constant defined in Proposition* 6.9.

We first show that, assuming Proposition 6.7 is true, Proposition 6.10 implies Proposition 6.4.

Proof of Proposition 6.4. Let A_0 be the set of those elements from A'' which are in an interval belonging to K. Let $\lambda = \delta/5$. By Proposition 6.10, there exists $\epsilon > 0$ such that

$$|A_0| < |A|s^{-(1-\epsilon)}s^{-\lambda}s = |A|s^{\epsilon-\lambda} \le |A|s^{-\lambda/2}.$$

Therefore, if we pick $0 < \eta < \lambda/2$, then all the elements of A_0 can be included among the exceptional $n/s^{1+\eta}$ elements of Proposition 6.4.

All the remaining elements, i.e., the elements of $A''\backslash A_0$, are now in intervals where the number of elements is smaller than $s^{1-\epsilon}$. Thus, again applying the algorithm based on Proposition 6.7, we put every element of A in the required place. (We assume that $\eta < \epsilon/2$.) To complete the proof, we must show that the elements of A'' can be moved in the manner described.

Let E be the set of edges of the graph G on the vertex set X associated with matrix $\boldsymbol{B}$ in Proposition 6.9. Property 1 of matrix $\boldsymbol{B}$ implies that the degrees of all the vertices are equal to $\ell = r^k$. We partition E into ℓ 1-factors and each interval I into s/ℓ classes of equal sizes. We associate each edge $e \in E$ connecting the intervals I and J with a pair of classes $I_e \in I$ and $J_e \in J$. Using the partition of E into 1-factors, we may define the pairs (I_e, J_e) so that each class occurs in exactly one pair associated with some edge e. Let ι_e be a one-to-one correspondence between the sets I_e and J_e. In each step, we swap the contents of the processors x and $\iota_e(x)$ for all $e \in E$ and $x \in I_e$. This can be done in a single step since each processor takes part in a single swap. The essential change is as follows. If exactly one processor in a pair $(x, \iota_e(x))$ contained an element of A, then this remains true, but the element from A will be in the other processor. (If either both or none of them contained an element of A, then this situation prevails.)

Originally, we only had intervals where the processors containing elements from A'' had either density at least $s^{-\epsilon}$ or density 0. Suppose that after performing a step described above, the set A'' will have density u_I in the interval I $(0 < u_I < 1)$. Let $\boldsymbol{w} = (w_I)_{I\in X}$ be the vector consisting of the original densities, and let $\boldsymbol{u} = (u_I)_{I\in X}$ be the vector of densities after one step has been performed. Our definitions imply that $\boldsymbol{u} = r^{-k}\boldsymbol{B}\boldsymbol{w}$. Let $\boldsymbol{v} = (v_I)_{I\in X}$ be the "characteristic" vector of $\boldsymbol{w}$, i.e.,

$$v_I = \begin{cases} 1 & \text{if } w_I \neq 0, \\ 0 & \text{if } w_I = 0. \end{cases}$$

By the nonnegativity of $\boldsymbol{B}$, $\boldsymbol{u}$, $\boldsymbol{v}$, and $\boldsymbol{w}$, we have $\|\boldsymbol{u}\| \le \|r^{-k}\boldsymbol{B}\boldsymbol{v}\|$. To give an upper bound on $\|\boldsymbol{u}\|$, we want to use property 3. The condition "k is sufficiently large" holds because of the assumptions $r^k > s^{1/8}$ and $s > s_0$. Also,

$$\sum v_I = |K| \le |A|/s^{1-\epsilon} = (n/s)/s^{1-\epsilon} \le |X|/s^{1-\epsilon}.$$

Since $m = |X|$ and $r^{2k} < s^{1/2} < s^{1-\epsilon}$, the requirements of property 3 in Proposition 6.9 are met. According to the conclusion there, we have $\|\boldsymbol{u}\| \leq r^{-\delta k}\|\boldsymbol{v}\| \leq r^{-\delta k}h^{1/2}$, where h is the number of nonzero components of $\boldsymbol{v}$. On the other hand, $\|\boldsymbol{u}\| \geq s^{-\epsilon}|K|^{1/2}$. Therefore, $s^{-\epsilon}|K|^{1/2} \leq r^{-\delta k}h^{1/2}$. Since $r \geq s^{1/8}$, we conclude that $|K| \leq s^{2\epsilon-\delta/4}h \leq s^{-\delta/5}h$. (Here we assumed that $\epsilon < \delta/20$.) □

We will need the following in the proof of Proposition 6.7.

PROPOSITION 6.11. *There is a positive c such that for every integer $s > 0$, if we have s processors, $s^{1/8} > |A|$, and the elements of A are at the first $s^{1/2}$ processors, then all the elements of A can be moved to the first $s^{1/4}$ processors in c steps.*

We will use the following well-known concepts in the proof. If K is a field, then the *affine plane* over K, $K \times K$, consists of all the ordered pairs of elements of K. A subset $L \subseteq K \times K$ is a line if there exist $a, b, c \in K$ such that $L = \{\langle x, y\rangle | ax + by + c = 0\}$. We say that two lines have the same direction if they are parallel, that is, they do not intersect. A direction is a maximal set of parallel lines.

Proof of Proposition 6.11. Suppose $p = s^{1/4}$ is a prime. (If $p = s^{1/4}$ is not a prime, then let p be an arbitrary prime between $\frac{1}{2}s^{1/4}$ and $s^{1/4}$.) We associate with each of the first $s^{1/2}$ processors a point in the affine plane with p^2 elements. There are $p = s^{1/4}$ directions (maximal sets of parallel lines) on the plane but less than $(s^{1/8})^2 = s^{1/4}$ pairs formed from the elements of A. Therefore, there is a direction such that each line of this direction contains at most one element from A. Since the number of processors is s, we can actually find such a direction in a constant number of steps. Indeed, we associate with each line e a processor P_e, and with each pair of points $\langle p_1, p_2\rangle$ ($p_1 \neq p_2$) a processor Q_{p_1,p_2}. If both points p_1 and p_2 contain an element of A, then the processor $Q(p_1, p_2)$ attempts to write in the register of processor P_e, where e is the line determined by the points p_1 and p_2. After this step, each processor P_e knows whether the line e contains more than one element of A, and in the next step, it will try in a similar way to transmit this information to processors associated with directions. Using the direction where each line contains at most one point from A, we may easily move A to the first $s^{1/4}$ processors. □

Proof of Proposition 6.7. If ϵ, ϵ', and c are positive, then denote by $\Psi(\epsilon, \epsilon', c)$ the following statement: for all positive integers s, if the elements of A ($|A| \leq s^{1-\epsilon}$) are given on s processors, then in c steps we can move all the elements to the first $s^{1-\epsilon'}$ processors.

Proposition 6.7 is the following assertion:

$$(\forall \epsilon > 0)\ (\exists \epsilon' > 0)\ (\exists c > 0)\ \Psi(\epsilon, \epsilon', c).$$

Let $\Theta(\epsilon)$ be the statement

$$(\exists \epsilon' > 0)\ (\exists c > 0)\ \Psi(\epsilon, \epsilon', c).$$

We have to prove that for all $\epsilon \in (0, 1)$, $\Theta(\epsilon)$.

We will present a strictly increasing continuous function $f : (0, 1) \to (0, 1)$ so that for each $\epsilon \in (0, 1)$, we have $\Theta(f(\epsilon)) \Rightarrow \Theta(\epsilon)$. Moreover, we will give an $\epsilon_0 \in (0, 1)$ with $\Theta(\epsilon_0)$. The existence of these objects implies Proposition 6.7 since the properties of the function f guarantee that for every $\epsilon > 0$ there is a positive integer i such that $f^{(i)}(\epsilon) > \epsilon_0$, where $f^{(i)}$ is the ith iterate of f. Therefore (using the fact that if $\alpha < \beta$, then $\Theta(\alpha) \Rightarrow \Theta(\beta)$), we have

$$\Theta(\epsilon_0) \Rightarrow \Theta(f^{(i)}(\epsilon)) \Rightarrow \Theta(f^{(i-1)}(\epsilon)) \Rightarrow \cdots \Rightarrow \Theta(\epsilon).$$

Now we prove $\Theta(\epsilon_0)$ with $\epsilon_0 = \frac{7}{8}$. More precisely, we prove $\Psi(\frac{7}{8}, \frac{1}{4}, c)$ for some c. Assume that $|A| \leq s^{1-\epsilon_0} = s^{1/8}$. Let Z be a set of $s^{1/2}$ elements. We associate each processor with an element of $Z \times Z$. Let

$$Y = \{y \in Z \mid (\exists z \in Z) \text{ (processor } \langle y, z \rangle \text{ stores an element of } A)\}.$$

We can actually find the elements of Y in a constant number of steps. Note that Y is represented as the set $Y \times \{z_0\}$ for an arbitrary $z_0 \in Z$. Also, $|Y| \leq |A| \leq s^{1/8}$. Let $\bar{A}$ be a new set (disjoint from A) whose elements have to be handled by the processors. Suppose that the elements of $\bar{A}$ are at the processors belonging to Y. We may assume that $Z \times \{z_0\}$ are the first $s^{1/2}$ processors, so the elements of $\bar{A}$ are at the first $s^{1/2}$ processors. Applying Proposition 6.11, we may move the elements of $\bar{A}$ to the first $s^{1/4}$ processors. We may also assume that at the end, each processor containing an $a \in \bar{A}$ also contains the address of the processor where a was initially. This implies that, coming back to the original set A, we may simultaneously move all the elements of A inside the sets $Z \times \{z\}$ (for each $z \in Z$) so that each element of A will move to a processor $\langle v, z \rangle$, where v is among the first $s^{1/4}$ elements of Z. Consequently, A is on the first $s^{3/4}$ processors, which concludes the proof of $\Theta(\epsilon_0)$. Thus we have reduced the proof of Proposition 6.7 to the question of existence of a function f with properties given in the following proposition. □

PROPOSITION 6.12. *There is a strictly increasing continuous function $f : (0,1) \to (0,1)$ such that for every $\epsilon \in (0,1)$, $\Theta(f(\epsilon)) \Rightarrow \Theta(\epsilon)$.*

For the proof of this proposition, we will need the following.

PROPOSITION 6.13. *If $\alpha > 0$ is sufficiently small, then for every $\xi > 0$, there is a positive c such that for all s, if $|A| = s^\alpha$ and the elements of A are on s processors, then they can be moved in c steps to the first $s^{\alpha+\xi}$ processors.*

Remark 6.14. We will use the following consequence of the proof of $\Theta(\epsilon_0)$: if $\alpha > 0$ is sufficiently small and $|A| \leq s^\alpha$, then we can move $|A|$ to the first $s^{3/4}$ processors in a constant number of steps. Iterating this step, we get the following: for all $\beta > 0$, if $\alpha > 0$ is sufficiently small, $|A| \leq s^\alpha$, and the set A is given on s processors, then we can move A to the first s^β processors in a constant number of steps.

Assume now that a sufficiently small $\beta > 0$ is fixed and $\alpha > 0$ is sufficiently small with respect to β. According to the previous remark, we may assume that A is on the first s^β processors. Let $m = s^\beta$. We now have the following situation: the elements of A are given on m processors, but we have $m^{1/\beta}$ extra processors that we can use for the computation. Therefore, we can simulate an unlimited-fan-in constant-depth Boolean circuit of size $m^{\beta/2}$ with m inputs. We associate the input nodes with the m processors. The value of the input will be 1 if there is an element of A at the processor and 0 otherwise. Using the following theorem (see [1]), we are able to approximately count the number of elements of A in a constant number of steps.

THEOREM 6.15. *There exist positive c and d such that for all positive integers n and $a \leq n$, there is an (explicitly constructed) unlimited-fan-in Boolean circuit C with n inputs and depth d and of size at most n^c so that for each input sequence $\boldsymbol{x}$, if $|\boldsymbol{x}|$ denotes the number of 1's in the sequence, then we have the following:*

1. *if $|\boldsymbol{x}| \leq (1 - (\log n)^{-1})a$, then $C(\boldsymbol{x}) = 0$, and*
2. *if $|\boldsymbol{x}| \geq (1 + (\log n)^{-1})a$, then $C(\boldsymbol{x}) = 1$.*

We will also use the following easy proposition from [1] about mod p polynomials.

DEFINITION 6.16. *Let p be a prime number and let K_p be a field with p elements. Assume that f is a polynomial of degree k with coefficients in K_p. We define a map*

$$h_f^p : K_p \times K_p \to K_p$$

as follows. If $\langle u, v\rangle \in K_p \times K_p$, then $h_f^p(\langle u, v\rangle) = u - f(v)$.

PROPOSITION 6.17. *If $X \subseteq K_p \times K_p$ and $|X| \le p^{1-2/(k+1)}$, then there exists a polynomial f of degree k with coefficients in K_p such that for all $y \in K_p$,*

$$|\{x \in K_p \times K_p \mid h_f^p(x) = y\}| \le k.$$

We use this proposition with $k = 4$, and we assume that $K_p \times K_p$ is the set of processors and X is the subset of processors where the elements of A are sitting. Proposition 6.17 for $k = 4$ implies that if $|A| < p^{3/5}$ and the elements of A are located in the first p^2 processor, then using p^{10} processors, we can move the elements of A to the first $4p$ processors in a constant number of steps. Since we have enough processors, we may check all of the polynomials of degree 4 simultaneously and find a polynomial f that satisfies the conclusions of the proposition. Using this polynomial, we can move the points in A to the first $4p$ processors. Iterating this step and using Remark 6.14, we get the following.

PROPOSITION 6.18. *If $\alpha > 0$ is sufficiently small and $|A| \le s^\alpha$, then using s processors, the elements of A can be moved to the first $s^{2\alpha}$ processors.*

Proof of Proposition 6.13. Assume now that $|A| \le s^\alpha$, where $\alpha > 0$ is sufficiently small. According to Proposition 6.18, we may assume that A is already in the first $|A|^2$ processors, and we may continue in the following way: assume that p is a prime with $|A| < p < 2|A|$. Since we are able to count approximately, we can find such a prime in a constant number of steps. (Since we can count approximately, we can find in a constant number of steps an interval which is contained in $(|A|, 2|A|)$ and is of length at least $|A|/2$. By the prime-number theorem, if $|A|$ is sufficiently large, such an interval always contains a prime.) We put the first p^2 processors on the affine plane with p^2 elements. There are only p^2 ordered pairs formed from the elements of A, and there are p directions (maximal set of parallel lines) on the plane. Therefore, there must be a direction such that the number of ordered pairs from A contained in the lines of this direction is at most p. Since $\alpha > 0$ is sufficiently small and A is on the first $s^{2\alpha}$ processors, we may actually find a direction (using approximate counting) where the number of ordered pairs is less than—say $2p$. Let $a_1, \ldots, a_p$ be the numbers of elements of A on the lines with this direction. We have $\sum_{i=1}^p a_i^2 \le 2p$.

Let $\mu > 0$ be sufficiently small. First, we consider the lines with $a_i \le p^\mu$. According to Proposition 6.18, within each line of this type, all of the elements of A can be moved to the first $p^{2\mu}$ processors. Therefore, every element of A contained in a line of this type can be moved to an array of size $pp^{2\mu} = 4s^{\alpha+2\mu} \le s^{\alpha+3\mu}$.

We claim that the number of elements of A on lines with $a_i > p^\mu$ is at most $p^{1-\mu}$. Indeed, the minimum of $\sum' a_i^2$ (where $\sum'$ stands for $\sum_{a_i > p^\mu}$) subject to $\sum' a_i =$ constant is attained when all the a_i's are equal, and in this case, we get the claimed result.

Thus in one step, we decreased the number of elements of A not in our array by a factor of p^μ. Continuing this process, in a constant number of steps, we will have all but $s^{\alpha/2}$ elements of A in an array of the required size. According to Proposition 6.18, the remaining $s^{\alpha/2}$ elements can be moved to an array of size s^α, which completes the proof of Proposition 6.13. □

Proof of Proposition 6.12. We first define f. Let $\alpha \in (0,1)$ so that Proposition 6.13 holds for α. For the remainder of the proof, we consider α as fixed. Let $f(\epsilon) = \epsilon(1+\alpha(1-\epsilon))$. Clearly, $f(\epsilon) \in (0,1)$ for all $\epsilon \in (0,1)$. Moreover, f is continuous and strictly increasing in this interval. Apart from these conditions, we will need only the following property of f: for all $\epsilon \in (0,1)$,

$$1-\epsilon-(1-\alpha)(1-f(\epsilon)) \leq \alpha - \alpha^2\epsilon.$$

Indeed,

$$\begin{aligned} 1-\epsilon-(1-\alpha)(1-f(\epsilon)) &= \alpha-\epsilon+\epsilon(1+\alpha(1-\epsilon))(1-\alpha) \\ &\leq \alpha-\epsilon+\epsilon(1+\alpha)(1-\alpha) = \alpha-\alpha^2\epsilon. \end{aligned}$$

Suppose $\epsilon \in (0,1)$ is fixed and $\Theta(f(\epsilon))$. We wish to show that $\Theta(\epsilon)$ holds. We divide the set of s processors into intervals of size $s^{1-\alpha}$. Let R be the set of all intervals of this type, and for all $\kappa > 0$, let $S_\kappa \subseteq R$ be the set of those intervals where the number of elements from A is at most $(s^{1-\alpha})^{1-\kappa}$. $\Theta(f(\epsilon))$ implies that for each $I \in S_{f(\epsilon)}$, the elements of A can be moved to the first $(s^{1-\alpha})^{1-\epsilon_1}$ processors in c_1 steps, where ϵ_1 and c_1 depend only on $f(\epsilon)$ (and thus only on ϵ.) Therefore, all of the elements of A contained in intervals of $S_{f(\epsilon)}$ can be moved to the first $s^\alpha(s^{1-\alpha})^{1-\epsilon_1} = s^{1-(\epsilon_1-\alpha\epsilon_1)}$ processors in a constant number of steps. Thus we have to consider only those elements of A that are contained in processors outside $S_{f(\epsilon)}$. (We first apply the algorithm described here to all the intervals and later work only with those elements of A that are in intervals where the algorithm did not work.)

Since each interval of $R \setminus S_{f_\epsilon}$ contains at least $(s^{1-\alpha})^{1-f(\epsilon)}$ elements of A and $|A| \leq s^{1-\epsilon}$, we have that $|R \setminus S_{f(\epsilon)}| \leq s^{1-\epsilon}/s^{(1-\alpha)(1-f(\epsilon))}$. This and the inequality proved after the definition of f imply that $|R \setminus S_{f(\epsilon)}| \leq s^{\alpha-\alpha^2\epsilon} = (s^\alpha)^{1-\alpha\epsilon}$. According to Proposition 6.13, using all of the s processors, we can construct a one-to-one mapping τ of $R \setminus S_{f(\epsilon)}$ into the set of the first $(s^\alpha)^{1-(\alpha\epsilon/2)}$ intervals of R. Moving all the elements of A from an interval I to the processors of the interval $\tau(I)$, we are able to move all of the elements of A to processors in the first $(s^\alpha)^{1-(\alpha\epsilon/2)}$ intervals. Thus we have moved the elements of A to the first $s^{1-\alpha}(s^\alpha)^{1-(\alpha\epsilon/2)} = s^{1-(\alpha\epsilon/2)}$ processors, and so $\Theta(\epsilon)$ holds with $\epsilon' = \alpha\epsilon/2$. This completes the proof of Proposition 6.12, which was the last step in the proof of Proposition 6.2. □

REFERENCES

[1] M. Ajtai, *Approximate counting with uniform constant depth circuits*, in Advances in Computational Complexity Theory, J.-Y. Cai, ed., DIMACS Series in Discrete Mathematics and Theoretical Computer Science 13, American Mathematical Society, Providence, RI, 1993, pp. 1–20.

[2] M. Ajtai, J. Komlós, W. L. Steiger, and E. Szemerédi, *Optimal parallel selection has complexity* $O(\log\log N)$, J. Comput. System Sci., 38 (1989), pp. 125–133.

[3] N. Alon and N. Megiddo, *Parallel linear programming in fixed dimension almost surely in constant time*, J. Assoc. Comput. Mach., 41 (1994), pp. 422–434.

[4] K. L. Clarkson, *Linear programming in* $O(n \times 3^{d^2})$ *time*, Inform. Process. Lett., 22 (1986), pp. 21–27.

[5] X. Deng, *An optimal parallel algorithm for linear programming in the plane*, Inform. Process. Lett., 35 (1990), pp. 213–217.

[6] D. Dobkin, R. J. Lipton, and S. Reiss, *Linear programming is* log *space hard for* P, Inform. Process. Lett., 8 (1979), pp. 96–97.

[7] M. E. DYER, *Linear time algorithms for two- and three-variable linear programs*, SIAM J. Comput., 13 (1984), pp. 31–45.

[8] ———, *On a multidimensional search technique and its application to the Euclidean one-center problem*, SIAM J. Comput., 15 (1986), pp. 725–738.

[9] O. GABBER AND Z. GALIL, *Explicit construction of linear-sized superconcentrators*, J. Comput. System Sci., 22 (1981), pp. 407–420.

[10] N. MEGIDDO, *Linear-time algorithms for linear programming in R^3 and related problems*, SIAM J. Comput., 12 (1983), pp. 759–776.

[11] ———, *Linear programming in linear time when the dimension is fixed*, J. Assoc. Comput. Mach., 31 (1984), pp. 114–127.

[12] ———, *Dynamic location problems*, Ann. Oper. Res., 6 (1986), pp. 313–319.

[13] R. M. TANNER, *Explicit concentrators from generalized N-gons*, SIAM J. Algebraic Discrete Meth., 5 (1984), pp. 287–293.

SIAM J. COMPUT.
Vol. 25, No. 6, pp. 1196–1230, December 1996

004

FEASIBLE TIME-OPTIMAL ALGORITHMS FOR BOOLEAN FUNCTIONS ON EXCLUSIVE-WRITE PARALLEL RANDOM-ACCESS MACHINES*

MARTIN DIETZFELBINGER†, MIROSŁAW KUTYŁOWSKI‡, AND RÜDIGER REISCHUK§

Abstract. It was shown some years ago that the computation time for many important Boolean functions of n arguments on concurrent-read exclusive-write parallel random-access machines (CREW PRAMs) of unlimited size is at least $\varphi(n) \approx 0.72 \log_2 n$. On the other hand, it is known that every Boolean function of n arguments can be computed in $\varphi(n)+1$ steps on a CREW PRAM with $n \cdot 2^{n-1}$ processors and memory cells. In the case of the OR of n bits, n processors and cells are sufficient. In this paper, it is shown that for many important functions, there are CREW PRAM algorithms that almost meet the lower bound in that they take $\varphi(n) + o(\log n)$ steps but use only a small number of processors and memory cells (in most cases, n). In addition, the cells only have to store binary words of bounded length (in most cases, length 1). We call such algorithms "feasible." The functions concerned include the following: the PARITY function and, more generally, all symmetric functions; a large class of Boolean formulas; some functions over non-Boolean domains $\{0, \ldots, k-1\}$ for small k, in particular, parallel-prefix sums; addition of n-bit numbers; and sorting n/l binary numbers of length l. Further, it is shown that Boolean circuits with fan-in 2, depth d, and size s can be evaluated by CREW PRAMs with fewer than s processors in $\varphi(2^d) + o(d) \approx 0.72d + o(d)$ steps. For the exclusive-read exclusive-write (EREW) PRAM model, a feasible algorithm is described that computes PARITY of n bits in $0.86 \log_2 n$ steps.

Key words. parallel random-access machine, exclusive-write, concurrent-read, exclusive-read, parallel time complexity, Boolean functions, Boolean formulas, Boolean circuits, symmetric functions, parallel prefix, parity, addition, sorting

AMS subject classifications. 68Q10, 68Q05, 68Q25

1. Introduction.

1.1. Motivation.

The parallel random-access machine (PRAM) is a powerful machine model that is often used for the design of parallel algorithms. Several variants of this model have been studied which differ in the rules that regulate concurrent access to memory cells in shared memory. In this paper, we concentrate on exclusive-write machines (CREW and EREW PRAMs), which do not allow concurrent-write access to shared memory cells. CREW PRAMs allow concurrent reading of one cell; EREW PRAMs do not. (Precise definitions of the models used will be given in §2. For a survey of PRAM models and algorithms for such models, see [15, 26].)

* Received by the editors December 30, 1991; accepted for publication (in revised form) February 14, 1995. A preliminary version of these results was presented at the 2nd Annual ACM Symposium on Parallel Algorithms and Architectures, July 1990 [11].

† Fachbereich Informatik, Universität Dortmund, D-44221 Dortmund, Germany (dietzf@ls2.informatik.uni-dortmund.de). The research of this author was partially supported by DFG grant ME 872/1-4 and the DFG-Forschergruppe "Effiziente Nutzung paralleler Systeme, Teilprojekt 4." This author was affiliated with the Universität-GH-Paderborn while this paper was written.

‡ Heinz Nixdorf Institut and Fachbereich Mathematik–Informatik, Universität-Gesamthochschule-Paderborn, D-33095 Paderborn, Germany (mirekk@uni-paderborn.de). The research of this author was partially supported by the Alexander von Humboldt-Stiftung (while visiting TH Darmstadt), Polish government program RP.I.09, DFG grant ME 872/1-4 (while visiting Paderborn), and KBN grants 2 1197 91 01 and 8 S503 002 07. This author was affiliated with the University of Wrocław while this paper was written.

§ Institut für Theoretische Informatik, Medizinische Universität zu Lübeck, Wallstraße 40, D-23560 Lübeck, Germany (reischuk@informatik.mu-luebeck.de). The research of this author was partially supported by the International Computer Science Institute, Berkeley, California. This author was affiliated with the Technische Hochschule Darmstadt while this paper was written.

The time complexity of Boolean functions on CREW PRAMs is quite well understood: using certain parameters corresponding to structural properties of such functions ("block-critical complexity" or "degree"; see below), it is possible to characterize their time complexity on CREW PRAMs up to a constant factor [22]. Methods were developed that allow proving lower bounds for the time complexity of many functions that are exact up to a small additive constant [9, 12, 17, 24].

In this context, for both upper and lower bounds, we use the standard definition of the "abstract CREW PRAM" from [9]. Each computation step consists of three phases, a READ, a COMPUTE, and a WRITE phase, which are executed synchronously by all processors. (For details, see §2.) This machine model abstracts from the cost of internal computations of the processors; the bounds hold regardless of the number of processors and the wordsize of the common memory cells. Such a strong model is perfectly acceptable for lower-bound proofs. Also, however, the general upper bounds that hold for Boolean functions on such machines are formulated with respect to the abstract PRAM. We consider two examples of such statements.

FACT 1.1. *Every Boolean function of n arguments can be computed in $\lceil \log n \rceil + 1$ steps*[1] *by an EREW PRAM with n processors and n memory cells of wordsize n.*

The algorithm behind this fact (in a binary-tree fashion, collect the whole input in one processor, which then computes and writes the output bit) is of limited value for concrete functions since it requires that the PRAM has a wordsize of n, i.e., the common memory cells store numbers of binary length up to n.

Before describing the second example, we introduce a function that plays a central role in the exact upper and lower bounds for computing Boolean functions on abstract CREW PRAMs. Let

$$\varphi(n) := \min\{j \mid F_{2j+1} \geq n\} \quad \text{for } n \geq 1,$$

where F_i denotes the ith Fibonacci number, i.e., $F_0 = 0$, and $F_1 = 1$, and $F_{i+2} = F_{i+1} + F_i$ for $i \in \mathbb{N}$. Note that from the well-known formula $F_i = (\Phi^i - \hat{\Phi}^i)/\sqrt{5}$ for $\Phi = \frac{1}{2}(1+\sqrt{5})$ and $\hat{\Phi} = \frac{1}{2}(1-\sqrt{5})$, it easily follows that for $b := \Phi^2 = \frac{1}{2}(3+\sqrt{5})$, we have

$$\log_b n \leq \varphi(n) \leq \log_b n + 1.34 \quad \text{for all } n \geq 1. \tag{1.1}$$

Also, note that $\log_b 2 \approx 0.72$ and hence $\varphi(n) \approx 0.72 \log n$.

Consider the function $\mathrm{OR}_n(x_1, \ldots, x_n) := x_1 \vee x_2 \vee \cdots \vee x_n$. It is known [9] that OR_n can be computed by an EREW PRAM with n processors and n memory cells of wordsize 1 in $\varphi(n)$ steps. Using this algorithm, it is easy to observe that all Boolean functions of n variables can be computed on CREW PRAMs almost as fast as OR_n (see [9]; also see [17]).

FACT 1.2. *Every Boolean function f of n arguments can be computed in $\varphi(n) + 1$ steps by a CREW PRAM with $n \cdot 2^{n-1}$ processors and $n \cdot 2^{n-1}$ memory cells of wordsize 1. Moreover, after step $\varphi(n) + 1$, there is a processor that knows all input bits.*

For many Boolean functions f, this fact corresponds to an algorithm with an optimal running time [12]. Again, however, this algorithm (essentially, for every possible input vector $a = (a_1, \ldots, a_n) \in f^{-1}(1)$, there is a team of n processors that checks whether the actual input $(x_1, \ldots, x_n)$ equals a and, if so, writes the result 1 to the output cell) is practically worthless because of the exponential number of processors required. Moreover, the algorithms to be carried out by the processors

[1] Throughout the paper, log stands for the logarithm with respect to base 2.

completely ignore any structure present in the function f: they simply represent the list of inputs in $f^{-1}(1)$. Algorithms of a similar kind are used in the general upper-bound proofs of [22].

Our focus in this paper is on algorithms for computing specific functions on exclusive-write PRAMs that are time optimal up to small *additive* terms and whose implementation is "feasible" in a sense discussed in the following. Specifically, we are interested in methods that do not use more than n processors and n common memory cells for computing functions of n arguments. For many interesting functions, there are "critical inputs" (see [9] and §1.2) for which each input bit must be read by some processor; this implies that $\Omega(n/\log n)$ processors are necessary if logarithmic computation time is to be achieved.

How strongly the hardware size may influence the parallel time complexity is shown, for example, by the PARITY function:

$$\mathrm{PARITY}_n(x_1, \ldots, x_n) := x_1 \oplus x_2 \oplus \cdots \oplus x_n \quad \text{for } n \geq 1.$$

On a concurrent-read *concurrent-write* (CRCW) PRAM with exponentially many processors and a common memory of exponential size, this function can be computed in two steps. However, with only n processors, the time complexity increases to $\Theta(\log n/\log\log n)$ [2]. A similar tradeoff holds with respect to the memory size. In general, we require that an n-processor machine has a common memory of size at least n. Otherwise, separate read-only input cells are necessary and the time complexity increases significantly due to the small communication bandwidth [31].

We may assume that the memory cells of a PRAM can store only binary numbers. Following [4], a PRAM is said to have wordsize w if the cells of its common memory can only hold binary words of length at most w. Although bounded wordsize may seem to be a reasonable requirement in combination with a processor bound of n, it is a severe restriction since for most Boolean functions of n arguments, the computation time is at least $n/w - o(n/w)$ on PRAMs with n processors and wordsize w [4]. The algorithms for special Boolean functions that will be presented in this paper require only constant wordsize; in most cases, wordsize 1 suffices. Note that in case the algorithms were implemented on a "concrete" PRAM with processors with local memories, this restriction on the wordsize would not apply to the memory cells or registers used by each processor in its local memory, which, e.g., have to hold addresses of common memory cells. These have to have wordsize at least logarithmic in the number of processors and global memory cells; this will also be sufficient for our algorithms.

Finally, we aim at simple programs for each processor. There seems to be no general agreement on what exactly this should mean, and we do not attempt to formalize this criterion; for a possible approach, see, for example, [29]. Nonetheless, we feel that the PRAM algorithms presented in this paper fulfill this condition for any reasonable definition of "simple program."

The fundamental example of a feasible, time-optimal CREW algorithm is the method for computing OR_n in $\varphi(n)$ steps mentioned above [9], which even works on an EREW PRAM. It is feasible since only n processors and n memory cells are used, all cells have wordsize 1, and the processors execute a very simple program. At first glance, it may seem that at least $\log n$ steps are necessary for computing OR_n. The essential observation that makes it possible to achieve the speedup from $\log n$ to $\varphi(n)$ is that in certain situations, a processor can transfer information into a common memory cell without destroying the information of that cell. This cannot be done by a direct WRITE to the cell since its prior contents would be overwritten; however, it can be achieved by *not writing.* This idea is exploited as follows in the algorithm for computing OR_n [9]: Assume that a processor P knows $y_1 \in \{0,1\}$ and a cell C stores a value $y_2 \in \{0,1\}$. By a single WRITE operation, C can be set to the value of $y_1 \vee y_2$. Namely, if $y_1 = 0$, then P does not have to write since $y_1 \vee y_2 = y_2$. If $y_1 = 1$, then P writes a 1 into C. Again, this gives the correct result since $y_1 \vee y_2 = 1 \vee y_2 = 1$. Using this trick, in one computation step (consisting of a READ, a COMPUTE, and a WRITE phase), the processors of a CREW PRAM can increase the amount of information stored in common memory cells by more than a factor of 2. As a consequence, it is possible to compute OR_n in fewer than $\log n$ steps.

1.2. Lower and upper bounds. Essentially, two general methods are known for proving lower bounds for the time complexity of Boolean functions on CREW PRAMs. The first is based on the concept of *critical complexity.* The critical complexity $c(f)$ of a function $f : \{0,1\}^n \to \{0,1\}$ is defined as the maximal number k for which there is an input $a = (a_1, a_2, \ldots, a_n)$ and a set $J \subseteq \{1, 2, \ldots, n\}$ of cardinality k such that

$$f(a) \neq f(a_1, \ldots, a_{i-1}, \bar{a}_i, a_{i+1}, \ldots, a_n) \quad \text{for all } i \in J.$$

A "critical input" a is one for which this condition is true with $J = \{1, \ldots, n\}$. A lower bound for CREW PRAMs on the basis of $c(f)$ has been proved in [9] (and improved in [24]).

FACT 1.3. *The time for a CREW PRAM to compute a function f is at least $\frac{1}{2}\log c(f)$, no matter how many processors and cells are used. Furthermore, the word-size may be arbitrarily large, and the computational power of the processors may be unlimited.*

A generalization of the critical complexity, the so-called *block-critical complexity* (or "block sensitivity"), $bc(f)$, was considered in [22], where it was shown that the lower bound of Fact 1.3 can be improved to $\frac{1}{2}\log bc(f)$ and that the time complexity of f on abstract CREW PRAMs is $O(bc(f))$.

The second general lower-bound method is based on an approach proposed in [17]. The idea is to consider the *degree* of a Boolean function when it is regarded as a polynomial over the integers. This useful complexity measure and variants thereof have applications far beyond complexity analysis of the CREW model; see, for example, [6, 23, 27, 30]. (The notion of "degree" of a Boolean function used in the lower-bound proof for CRCW PRAMs of [2] is different.) Each Boolean function f can be represented by a polynomial over $x_1, \ldots, x_n$ with coefficients from $\mathbb{Z}$. This representation is unique if each x_i^d for $d > 1$ and $1 \leq i \leq n$ is reduced to x_i. The degree of the polynomial representing f is called the degree of f and is denoted by $\deg(f)$. In [12], the authors have shown the following.

FACT 1.4. *At least $\varphi(\deg(f))$ steps are required for computing a Boolean function f on an arbitrarily large and powerful CREW PRAM.*

For the function OR_n, this gives the lower time bound $\varphi(n)$, which, in view of the upper bound obtained in [9], is the best possible. The lower bounds based on critical complexity and on degree complexity are not identical in the sense that there are functions f such that $\frac{1}{2}\log(bc(f))$ is smaller by a constant factor than $\varphi(\deg(f)) \approx 0.72\log(\deg(f))$ and vice versa. However, $\deg(f)$ and $bc(f)$ are polynomially related [12, 22, 30]; hence these two lower bounds differ from each other and from the CREW complexity of f at most by a constant factor.

There are Boolean functions that nontrivially depend on n arguments and can still be computed in $o(\log n)$ steps on a CREW PRAM. However, for almost all functions of n arguments, we have $c(f) \geq n - 1$ [7], which implies a lower bound $\frac{1}{2}\log(n-1)$. One can even show that almost all functions of n arguments have degree n [12] and hence cannot be computed in fewer than $\varphi(n)$ steps. On the other hand, we have the upper bound $\varphi(n) + 1$ noted in Fact 1.2, which holds for all Boolean functions.

1.3. Results. In this paper, we will consider the following general problem:

Let a Boolean function f of n arguments with $\deg(f) = \Omega(n)$ *be given. Design a feasible CREW PRAM algorithm that computes f in $\varphi(n) + o(\log n)$ steps, i.e., in almost optimal time.*

In general, by the results in [4], such algorithms do not exist. For most Boolean functions of n arguments, if the number of processors is polynomially bounded in n and the wordsize is bounded by $o(n/\log n)$, then the CREW complexity becomes much larger than $\log n$. Nonetheless, we will construct feasible algorithms for many important and natural functions.

First, we consider a variant of the aforementioned task for EREW PRAMs, which seems to be substantially harder than for CREW PRAMs, due to the following observations. No lower bounds for EREW PRAMs are known that would not be valid for CREW PRAMs as well. On the other hand, most known CREW PRAM algorithms use the concurrent-read operation in a substantial way. Moreover, no fast simulation of the concurrent-read operation on EREW PRAMs is possible [3], which makes it necessary to design efficient EREW algorithms by completely different techniques than for CREW algorithms. There are some results elaborating on what EREW PRAMs can do less efficiently than CREW PRAMs [3, 14, 28], but a deep understanding of the EREW PRAM model is still missing. We will construct a feasible algorithm for the EREW PRAM that computes PARITY$_n$ in approximately $0.86 \log n$ steps (Theorem 3.1). This is the second example—after the algorithm for OR$_n$—of an EREW algorithm with time complexity below $\log n$ for a function of degree n.

For the CREW PRAM model we will obtain the following results with respect to PARITY$_n$:

- PARITY$_n$ can be computed in $\varphi(n) + 1$ steps with $2^{O(\log^2 n)}$ processors and cells of wordsize 1, i.e., with significantly less than exponential hardware size (Theorem 4.1).
- PARITY$_n$ can be computed in time $\varphi(n) + O(\sqrt{\log n}\,)$ with n processors and n cells of wordsize 1 (Theorem 5.1).

Computing OR$_n$ or PARITY$_n$ amounts to evaluating a formula built with an associative operator over a 2-valued domain. We may generalize the results for these special functions to Boolean functions that are described by formulas or circuits:

- Any Boolean circuit of depth d and size s (with gates of fan-in 2) can be evaluated in $\varphi(2^d) + O(d/\log\log d) = 0.72...d + o(d)$ steps by a CREW PRAM with $o(s)$ processors (Theorem 5.5); if the circuit is a formula (i.e., the gates have fan-out 1), then $2^d/\log d$ processors can evaluate it in time $\varphi(2^d) + O(d/\log d)$ (Theorem 5.6).

The results just mentioned can further be generalized to formulas $F(x_1, \ldots, x_n) = x_1 \otimes \cdots \otimes x_n$, where $\otimes$ is an associative binary operator over a k-valued domain, say, $\{0, 1, \ldots, k-1\}$ for some $k \geq 2$:

- Any such F can be computed in $\varphi(n) + 1$ steps on a CREW PRAM with $2^{O(\log^2 n \cdot \log k)}$ processors and cells of wordsize 1, with the obvious exception of the cells storing the input and the output (Theorem 6.1).

- Any such F can be computed in $\varphi(n) + O(\sqrt{\log n \cdot \log k})$ steps on a CREW PRAM with n processors and n cells of wordsize $\|k-1\|$, where $\|r\|$ denotes the number of digits in the binary representation of r, i.e., $\|r\| = \lceil \log(r+1) \rceil$ for integers $r > 0$ (Theorem 6.4).

The results concerning Boolean circuits and formulas may also be generalized to arbitrary circuits and formulas over k-valued domains (Theorems 6.5 and 6.6).

Further, we develop a method for computing in parallel all prefix products $x_1 \otimes \cdots \otimes x_i$ of $x_1 \otimes x_2 \otimes \cdots \otimes x_n$ for an arbitrary associative operator $\otimes$ over a k-valued domain. The complexity bounds are the same as those for computing only the product (Theorem 7.3). The parallel-prefix operation has many applications; we consider just one of the most important ones:

- An n-processor CREW PRAM with a common memory of size n and wordsize 2 can add two binary numbers of length n in time $\varphi(n) + O(\sqrt{\log n})$.

In §8, we will present feasible algorithms with almost optimal running time for symmetric functions:

- Every symmetric Boolean function of n variables can be computed in $\varphi(n) + O(\log^{2/3} n \cdot \log\log n)$ steps by an n-processor CREW PRAM with n memory cells of wordsize 1 (Corollary 8.9).

The methods developed for symmetric functions can be used to show that also the problem of sorting bits or numbers can be solved by feasible algorithms in almost optimal time:

- n bits can be sorted in time $\varphi(n) + O(\log^{2/3} n \cdot \log\log n)$ by an n-processor CREW PRAM with n memory cells of wordsize 1 (Theorem 9.1).
- A CREW PRAM with $m^2 \cdot k$ processors can sort m binary numbers of length k in time $\varphi(m \cdot k) + O(\log^{2/3} m \cdot \log\log m)$ using $m \cdot (m+1) \cdot k$ memory cells of wordsize 1 (Theorem 9.3).

Remark 1.5. Theorem 5.5 (mentioned above) provides a general way for obtaining a fast, feasible CREW PRAM algorithm for a Boolean function f of n variables as follows:

1. Design a circuit (with gates of fan-in 2) for f with depth d as small as possible and size $s = O(n)$;
2. evaluate the circuit via the algorithm given in Theorem 5.5.

(Recall that if f is nondegenerate, i.e., f depends on all n variables, we must have $d \geq \log n$.) In some cases, it will even be possible to describe f by a formula of small depth over a 2-valued or a k-valued domain for small k and thus benefit from Theorem 5.6 or 6.6 (mentioned above). Note that step 1 has already been carried out for many functions, so results from the literature can be used. This approach will yield close-to-optimal time bounds ($\varphi(n) + o(\log n)$) with a small number of processors ($o(n)$) for some functions treated in this paper, such as OR_n, PARITY_n, or the addition of two binary numbers of n bits. However, in all of these cases, the additional $o(\log n)$ term is much smaller in the direct approach than in the algorithm resulting from the general method. For some other functions, e.g., the symmetric functions or the sorting function, no circuits are known whose simulation would yield a feasible algorithm with a running time of $\varphi(n) + o(\log n)$ since the best linear-size circuits known have depth $(1 + \Omega(1)) \log n$. The most drastic example of the failure of this approach to constructing fast CREW PRAM algorithms is provided by the well-known storage-access function

$$\mathrm{SA}_k \colon \{0,1\}^{k+2^k} \ni (y_{k-1}, \ldots, y_0, x_0, \ldots, x_{2^k-1}) \mapsto x_{(y_{k-1} \cdots y_0)_2} \in \{0,1\},$$

where $(y_{k-1} \cdots y_0)_2 \in \mathbb{N}$ is the number with binary representation $y_{k-1} \cdots y_0$. The minimum depth of a circuit for this function is $k + \log k + \Theta(1)$ [33, p. 78], but there is a CREW PRAM algorithm for this function that uses 2^k processors, i.e., fewer than the number of variables, and has running time $\varphi(k) + O(1)$, i.e., only logarithmic in the depth of the circuit (use Lemma 8.8 below).

2. Preliminaries.

We recall the definition of abstract CREW and EREW PRAMs (cf. [9]). A PRAM consists of some number of processors $P_1, P_2, \ldots$ and common memory cells $C_1, C_2, \ldots$ which can be read from and written to by each of the processors. The computation proceeds in steps; each step consists of three phases. During the first phase, a processor may read from a memory cell (the READ phase); during the second phase, a processor changes its internal state according to the information read; during the third phase, a processor may write into one memory cell (the WRITE phase). More precisely, we can describe the way in which a PRAM M works as follows. Let Q be the set of internal states of the processors of M and Σ be the set of symbols that can be written into the memory cells. Associated with each processor P_i of M, there are an initial state $q_i^0 \in Q$, a read-address function $\rho_i : Q \to \mathbb{N}$, a state-transition function $\delta_i : Q \times (\Sigma \cup \{\$\}) \to Q$ for $\$ \notin \Sigma$, a write-address function $\tau_i : Q \to \mathbb{N}$, and a write-value function $\sigma_i : Q \to \Sigma$.

During a single step, if processor P_i is in state q, then it reads from cell C_j, where $j = \rho_i(q) \geq 1$; it does not read if $\rho_i(q) = 0$. If P_i reads a symbol $u \in \Sigma$, then the state of P_i changes to $q' = \delta_i(q, u)$. If P_i has decided not to read, then P_i changes its state to $q' = \delta_i(q, \$)$. During the third phase, P_i writes a symbol $v = \sigma_i(q')$ into cell $C_{j'}$, where $j' = \tau_i(q')$ if $\tau_i(q') > 0$, and does not write if $\tau_i(q') = 0$.

A PRAM M is a CREW PRAM if for no admissible initial set of values stored in the memory cells it happens that during some step in the computation of M, two or more processors of M write into the same memory cell. In other words, M is a CREW PRAM if no write conflicts occur during a computation of M. A CREW PRAM M is an EREW PRAM if, additionally, two processors never *read* from the same cell during any step.

We say that a PRAM M computes a function f in T steps if the following holds. Initially, the arguments of f are stored in some fixed memory cells of M, each argument in a separate cell. After executing at most T steps, the value of f on the given arguments is stored in one or several dedicated memory cells and all processors have stopped their computations. Here we are mainly interested in computing Boolean functions, that is, functions f of the form $f : \{0,1\}^n \to \{0,1\}^m$ for n, $m \in \mathbb{N}$. A PRAM is said to have wordsize bounded by w for some $w \geq 1$ if $|\Sigma| \leq 2^w$.

3. Fast computation of PARITY on EREW PRAMs.

This section deals with the EREW model. Our goal is to show that, like the OR, the PARITY of n bits can be computed in fewer than $\log n$ steps. The key to the method is a way for computing the PARITY of five bits in just two steps. Two technical tricks are necessary to achieve this goal. The first one is useful for CREW PRAMs, too, and will be exploited again in later sections. The second one is specific for EREW PRAMs and is one of the basic elements of a time-optimal feasible broadcasting EREW algorithm presented in [3].

The first trick. If an algorithm wants to compute the PARITY of n bits in fewer than $\log n$ steps, it must in a single writing phase combine information known to a processor with some other information stored in a memory cell. This is not possible by direct writing since the old content of the memory cell would be overwritten. As in the fast OR algorithm of [9], processors will transmit the information by *not writing.*

The details are as follows. Let processors P_0 and P_1 know a Boolean value y_1 and cells C_0 and C_1 store a Boolean value y_2. (We say a processor P_i *knows* a value y at the end of step t if this value is a function of the state q of processor P_i at the end of step t. One may imagine that part of the state of P_i is a register that explicitly contains y.) In order to leave information in some cell that is sufficient to determine $y_1 \oplus y_2$, the following instruction is executed in parallel for $i = 0, 1$ (see Figure 3.1):

- P_i writes $*$ into cell C_i if and only if $y_1 \neq i$.

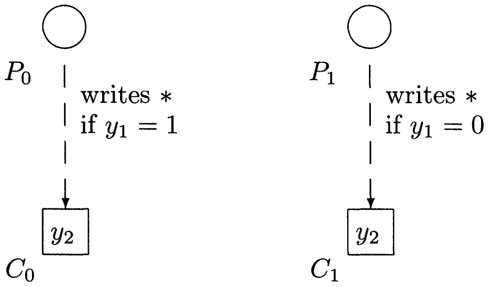


FIG. 3.1. *The first trick: combining information by not writing.*

Afterwards, one memory cell contains $*$ while the other remains unchanged. Suppose that $y_1 = 0$. Then C_1 is set to $*$ and C_0 still contains y_2. If a processor reads C_0, it encounters a symbol z different from $*$, and can conclude that y_1 must be equal to 0. Hence, it can deduce that $y_1 \oplus y_2$ is equal to z. If $y_1 = 1$, then C_0 is set to $*$ while y_2 remains in C_1. If a processor reads z in C_1, it can deduce that $y_1 \oplus y_2 = 1 \oplus z$. In both cases, the processor that has read (C_0 or C_1, respectively) knows $y_1 \oplus y_2$.

The second trick. In order to use the first trick, two different memory cells are needed that store the same value. We show how in one step a single processor can "write" a single value into two different cells (see Figure 3.2). Suppose that a processor P has to write $y \in \{0, 1\}$ into cells C_0 and C_1. The cells C_0 and C_1 are prepared in advance so that C_i contains i. If $y = 0$, then it suffices to change the contents of C_1 from 1 to 0; otherwise, C_0 has to be corrected. Hence in each case, it suffices that processor P writes y into the cell C_{1-y}. In the algorithm below, the preparation of C_0 and C_1 will be done by free processors immediately before the writing of P occurs. Thus no additional time is needed for the preprocessing.

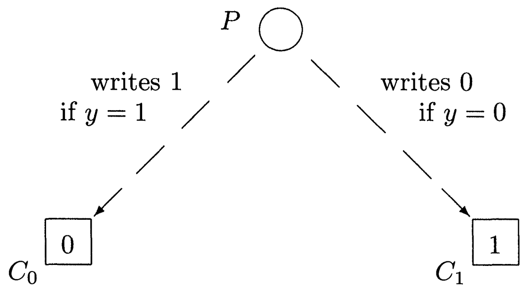


FIG. 3.2. *The second trick: writing two identical bits in one step.*

THEOREM 3.1. *For each n, the function* PARITY$_n$ *can be computed by an n-*

processor EREW PRAM with $2n$ *memory cells of wordsize* 1 *in time*

$$2 + 2\lceil \log_5 (n/2) \rceil \approx 0.86 \cdot \log n.$$

Proof. For the sake of simplicity, we assume that $n = 2 \cdot 5^k$ for some $k \in \mathbb{N}$. In the first step of the EREW algorithm and the READ phase of its second step, each input x_l, $1 \leq l \leq n$, is copied to an additional cell (say x_l from input cell C_l to C_{n+l}) and for each j, $1 \leq j \leq n/2$, two processors (say P_{2j-1} and P_{2j}) compute $\text{PARITY}(x_{2j-1}, x_{2j})$.

After this preprocessing, the main procedure starts. It consists of k stages, where each stage comprises four phases of alternating WRITEs and READs, starting with a WRITE. Thus a stage is made up of the last part of an EREW step (a WRITE), a complete step, and the first part of another step (a READ).

The input for stage i for $i = 1, \dots, k$ is described by a sequence of Boolean values $y_1^i, y_2^i, \dots, y_{n_i}^i$, where $n_i = n/5^{i-1}$, such that

$$\text{PARITY}(x_1, \dots, x_n) = \text{PARITY}(y_1^i, y_2^i, \dots, y_{n_i}^i).$$

The following properties are demanded at the beginning of each stage i:

- For each $1 \leq l \leq n_i$, there exist two cells that contain the value y_l^i.
- For every $1 \leq j \leq n_i/2$, two processors know $\text{PARITY}(y_{2j-1}^i, y_{2j}^i)$.

Assume that these properties hold at the beginning of stage i. Divide $y_1^i, y_2^i, \dots, y_{n_i}^i$ into groups of ten elements each:

$$\{y_1^i, \dots, y_{10}^i\},\ \{y_{11}^i, \dots, y_{20}^i\}, \dots .$$

For each j, to get the cells that store y_{2j-1}^{i+1} and y_{2j}^{i+1} and the processors that know $\text{PARITY}(y_{2j-1}^{i+1}, y_{2j}^{i+1})$, the machine uses only the cells and processors associated with the jth group. Since the computation for each group is essentially the same, we describe only how y_1^{i+1}, y_2^{i+1}, and $\text{PARITY}(y_1^{i+1}, y_2^{i+1})$ are computed. To simplify the description, processors and cells are named as follows. At the beginning of stage i, let

processors P_1 and P_2	know	$\text{PARITY}(y_1^i, y_2^i)$,
processors P_3 and P_4	know	$\text{PARITY}(y_3^i, y_4^i)$,
processors P_7 and P_8	know	$\text{PARITY}(y_7^i, y_8^i)$,
processors P_9 and P_{10}	know	$\text{PARITY}(y_9^i, y_{10}^i)$,
cells C_5 and C_5'	contain	y_5^i, and
cells C_6 and C_6'	contain	y_6^i.

During stage i, processors P_1 through P_{10} act as follows (see Figure 3.3).

First WRITE. During the first WRITE, the information known to processors P_1 and P_2 is combined with the information stored by cells C_5 and C_5' (and the information known by P_7 and P_8 with that stored in C_6 and C_6') using the first trick described above:

- If $\text{PARITY}(y_1^i, y_2^i) = 1$, then P_1 writes $*$ into C_5.
- If $\text{PARITY}(y_1^i, y_2^i) = 0$, then P_2 writes $*$ into C_5'.
- If $\text{PARITY}(y_7^i, y_8^i) = 1$, then P_7 writes $*$ into C_6.
- If $\text{PARITY}(y_7^i, y_8^i) = 0$, then P_8 writes $*$ into C_6'.

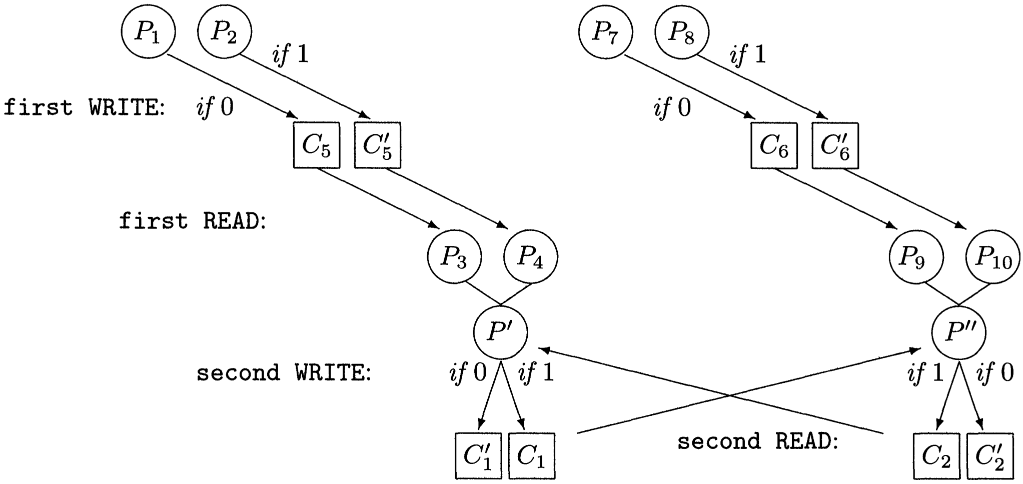


FIG. 3.3. *Flow of information within a stage.*

First READ. Since the other processors do not know which of the cells C_5 and C_5' contain the full information about PARITY(y_1^i, y_2^i, y_5^i), we use two processors to read both cells in parallel:

- Processor P_3 reads C_5 and processor P_4 reads C_5'.

One of them encounters $*$ and terminates its work. The other one—call it P'—using the information read and its own knowledge of PARITY(y_3^i, y_4^i), determines

$$\text{PARITY}(y_1^i, y_2^i, y_3^i, y_4^i, y_5^i) = y_1^{i+1}.$$

- Processor P_9 reads C_6 and processor P_{10} reads cell C_6'.

Similarly, one of them halts while the other one—call it P''—computes

$$\text{PARITY}(y_6^i, y_7^i, y_8^i, y_9^i, y_{10}^i) = y_2^{i+1}.$$

Second WRITE. The purpose of this writing round is that P' writes y_1^{i+1} into some cells C_1 and C_1' and P'' writes y_2^{i+1} into some cells C_2 and C_2'. For this, we apply the second trick described above. Thus the cells C_1, C_1', C_2, and C_2' must be prepared so that C_1 and C_2 contain 0 and C_1' and C_2' contain 1 (this can be done by processors P_3, P_4, P_9, and P_{10} during the first WRITE). Then

- P' writes 1 into cell C_1 if $y_1^{i+1} = 1$ and writes 0 into C_1' if $y_1^{i+1} = 0$,
- P'' writes 1 into cell C_2 if $y_2^{i+1} = 1$ and writes 0 into C_2' if $y_2^{i+1} = 0$.

Second READ. • Processor P' reads cell C_2 and processor P'' reads cell C_1.

Now, knowing y_1^{i+1} and y_2^{i+1}, both can compute PARITY(y_1^{i+1}, y_2^{i+1}).

After performing stage i, there will be two cells containing y_1^{i+1} (cells C_1 and C_1'), two cells containing y_2^{i+1} (cells C_2 and C_2'), and two processors (P' and P'') knowing PARITY(y_1^{i+1}, y_2^{i+1}), as required.

Note that during this procedure, concurrent READ or WRITE operations do not occur as long as each pair of processors (P_1, P_2), (P_3, P_4), (P_7, P_8), and (P_9, P_{10}) agree on their specific roles. Assuming that this is guaranteed for stage i, it can also be achieved for the next stage by declaring P' (the unique "survivor" of P_3 and P_4) as first. Also, stages can easily be lined up since the communication between stages is through fixed memory cells.

For $n = 2 \cdot 5^k$, the algorithm uses k stages to compute $\text{PARITY}(y_1^k, y_2^k) = \text{PARITY}(x_1, \dots, x_n)$. In an extra writing phase, the first processor of the two processors knowing $\text{PARITY}(y_1^k, y_2^k)$ writes the result into the output cell. The total number of steps is equal to

$$\tfrac{3}{2} + 2k + \tfrac{1}{2} = 2 + 2 \cdot \lceil \log_5 (n/2) \rceil \approx 0.86 \cdot \log n. \qquad \square$$

Remark 3.2. The upper time bound for computing PARITY on EREW PRAMs presented above is not optimal. One can modify the algorithm to get a slightly smaller computation time. However, the construction is much more complicated and therefore will be omitted.

4. A time-optimal CREW algorithm for PARITY with subexponentially many processors. Time bounds for computing the PARITY function on machines with a bounded number of processors have been studied extensively. For the most powerful PRAM model, the CRCW PRAM, in [2], an optimal lower bound of order $\log n / \log\log n$ is proved for the case where the number of processors is bounded by a polynomial. In [12], it has been shown that computing PARITY_n on a CREW PRAM takes at least $\varphi(n)$ steps (and, for some n, even $\varphi(n)+1$; see [18]) regardless of the number of processors. In this section, we will prove that the time bound $\varphi(n)+1$ can be achieved for PARITY_n with much less than exponential hardware size. The algorithm described in this section is not feasible in the sense discussed in §1 because it uses too many processors and cells and hence too large addresses as well. However, in the next section, the algorithm will be used as a component of a feasible algorithm for PARITY_n that is almost time optimal.

THEOREM 4.1. *Let $n = F_{2t-1}$. Then* PARITY_n *can be computed by a p-processor CREW PRAM M with m common memory cells of wordsize* 1 *in $t = 1 + \varphi(n)$ steps, where $p = n \cdot 2^{(t+1)t/2} \approx n \cdot 2^{(0.26 \log^2 n)}$ and $m = n \cdot (2^{t-1} + 1) \approx n^{1.72}$.*

Proof. The only property of PARITY_n used in the construction below is that this function can be computed by an iteration of an associative binary operator. This makes it possible to generalize the construction for PARITY_n to any associative operator (see Theorem 6.1), but here we will describe only the simple case of PARITY_n.

The CREW PRAM M constructed below combines the method used in [9] to compute the logical OR with the technique used by the fast EREW PRAM algorithm of §3. Since the algorithm is quite involved, we will describe it incrementally, each time giving more technical details. In order to achieve computation time $\varphi(n)+1$, during each WRITE, the processors must combine their knowledge with the information stored in the memory cells. This is not possible by direct writing, that is, by overwriting; instead, we will use a variation of the first trick from the previous section.

There are n groups of $2^{(t+1)t/2}$ processors and n groups of 2^{t-1} memory cells. (Each group corresponds to a single processor or a cell from the algorithm of [9] for computing OR_n.) During the computation, each processor and each cell is either active or dead. (Dead cells are those that contain the symbol $*$.) Initially, all processors and cells are active. Once a cell or a processor becomes dead, it remains in this state until the end of the computation. At any moment, all active processors of a group have the same knowledge and all active memory cells of a group code the same information about the input string. Which processors and cells are active depends on the current time step and on the input. A processor changes from the active to the dead state if it reads a dead cell; a cell changes to the dead state when $*$ is written into it. During the computation, more and more processors and cells die, but for each group, there is always a processor or a cell that is still active.

Each of the 2^{t-1} memory cells in a group has a *name*, which is a binary string of length $t-1$. The information carried by an active cell is not its content, but its name: the names of all active cells of a group agree in a prefix of a certain length, and this prefix codes all information that the cells have about the input string. We briefly explain how this works. Let $\mathcal{C}$ be one of the groups of cells. During the first WRITE, there is a group of processors that has to send a value $y_1 \in \{0,1\}$ to the memory cells of group $\mathcal{C}$.

- If $y_1 = 0$, then each cell of $\mathcal{C}$ with a name starting with 1 receives $*$; the cells with names starting with 0 remain unchanged.
- If $y_1 = 1$, then the roles are reversed: the cells in $\mathcal{C}$ with names starting with 0 are "killed"; the other cells remain active.

During the second WRITE, a different group of processors has to send a value $y_2 \in \{0,1\}$ to the cells of $\mathcal{C}$.

- If $y_2 = 0$, all cells in $\mathcal{C}$ whose names have a 1 in the second position receive $*$.
- If $y_2 = 1$, all cells in $\mathcal{C}$ whose names have a 0 in the second position receive $*$.

Obviously, the names of those cells in $\mathcal{C}$ that "survive" both WRITEs start with bits y_1 and y_2. Similarly, during step s, all cells of group $\mathcal{C}$ are killed that have names whose sth bit differs from some y_s. In that way, after step s, all active cells of $\mathcal{C}$ have names that agree in a prefix of length s. Thus if a cell of $\mathcal{C}$ with a name $u_1u_2\cdots u_{t-1}$ does not contain $*$ after step s, then it carries the information that $y_1 = u_1, y_2 = u_2, \ldots, y_s = u_s$, where $y_1, y_2, \ldots, y_s$ are the bits representing the information transmitted to $\mathcal{C}$ by the processors. In that sense, at each step, information "written" by processors does not overwrite information carried by cells.

Why do we need so many processors in each group? During a single READ, the active processors of one group read the memory cells of some other group. It would be best to read only the active cells, but since it is impossible to foresee which of the cells are active, the active processors are distributed evenly among the cells of the group and read all of them. Most of the processors die because they read dead cells, but a small fraction always survives. Because this fraction gets smaller with each step, the number of processors in each group has to be much larger than the number of memory cells.

We now describe the basic properties of the flow of information during the computation of M. For each $i \leq n$, during the first step, M writes $*$ into each cell of group i whose name has the bit $\bar{x}_i$ in its first position. In this way, x_i is coded by the memory cells of group i. Afterwards, the cell contents are not changed except that more and more cells die (receive a $*$). For each $i, j \leq n$, after step k,

- all active memory cells of group i code $\text{PARITY}(x_i, x_{i+1}, \ldots, x_{i+F_{2k}-1})$,
- all active processors of group j know $\text{PARITY}(x_j, x_{j+1}, \ldots, x_{j+F_{2k-1}-1})$

(compare with the algorithm in [9] for OR_n). It may happen that $i + F_{2k} - 1 > n$ or $j + F_{2k-1} - 1 > n$. Therefore, for $l > n$, by x_l we mean $x_{l'}$, where $l' = l \bmod n$. Similarly, when we talk about a group i of processors (cells) and $i \notin \{1, \ldots, n\}$, then we mean a group i' for $i' = i \bmod n$.

Now let us describe what happens during step $k+1$. The active processors of group j read from the cells of group $j + F_{2k-1}$. A lot of processors encounter dead cells and terminate their work. Each of the active processors can deduce the value

$$\mathrm{PARITY}(x_{j+F_{2k-1}}, \ldots, x_{j+F_{2k-1}+F_{2k}-1}) = \mathrm{PARITY}(x_{j+F_{2k-1}}, \ldots, x_{j+F_{2k+1}-1})$$

from the address of the cell it reads; hence it can compute

$$\begin{aligned}\mathrm{PARITY}(x_j, \ldots, x_{j+F_{2k-1}-1}) &\oplus \mathrm{PARITY}(x_{j+F_{2k-1}}, \ldots, x_{j+F_{2k+1}-1}) \\ &= \mathrm{PARITY}(x_j, x_{j+1}, \ldots, x_{j+F_{2k+1}-1}) \\ &= \mathrm{PARITY}(x_j, x_{j+1}, \ldots, x_{j+F_{2(k+1)-1}-1}),\end{aligned}$$

which is the value that the processors of group j have to know after step $k+1$. During the WRITE phase of step $k+1$, the active processors of group j write into some cells of group $j - F_{2k}$. The way of writing depends on the value

$$s = \mathrm{PARITY}(x_j, x_{j+1}, \ldots, x_{j+F_{2k+1}-1})$$

known to the processors. The symbol $*$ is written into each cell of group $j - F_{2k}$ with a name whose $(k+1)$st bit differs from s. Recall that the first k bits of the name of each cell that are still active before step $k+1$ code

$$\begin{aligned}&\mathrm{PARITY}(x_{j-F_{2k}}, x_{j-F_{2k}+1}, \ldots, x_{j-F_{2k}+F_{2k}-1}) \\ &\quad = \mathrm{PARITY}(x_{j-F_{2k}}, x_{j-F_{2k}+1}, \ldots, x_{j-1}).\end{aligned}$$

On the other hand, bit $k+1$ of the name of the cells that are active after step $k+1$ is equal to s, hence each such cell now codes

$$\begin{aligned}\mathrm{PARITY}&(x_{j-F_{2k}}, x_{j-F_{2k}+1}, \ldots, x_{j-1}) \oplus s \\ &= \mathrm{PARITY}(x_{j-F_{2k}}, \ldots, x_{j-1}) \oplus \mathrm{PARITY}(x_j, \ldots, x_{j+F_{2k+1}-1}) \\ &= \mathrm{PARITY}(x_{j-F_{2k}}, \ldots, x_{(j-F_{2k})+(F_{2k}+F_{2k+1}-1)}) \\ &= \mathrm{PARITY}(x_{j-F_{2k}}, \ldots, x_{(j-F_{2k})+F_{2(k+1)}-1}).\end{aligned}$$

Note that this is the value that must be coded by the active cells of group $j - F_{2k}$ after step $k+1$. After step $t-1$, the remaining active processors of the first group know

$$\mathrm{PARITY}(x_1, x_2, \ldots, x_{F_{2t-3}}).$$

They try to determine the value

$$\mathrm{PARITY}(x_{F_{2t-3}+1}, \ldots, x_{F_{2t-3}+F_{2t-2}}) = \mathrm{PARITY}(x_{F_{2t-3}+1}, \ldots, x_{F_{2t-1}})$$

by reading from the remaining active cell of group $F_{2t-3}+1$. One of them survives the reading and computes

$$\begin{aligned}\mathrm{PARITY}(x_1, x_2, \ldots, x_{F_{2t-3}}) &\oplus \mathrm{PARITY}(x_{F_{2t-3}+1}, \ldots, x_{F_{2t-1}}) \\ &= \mathrm{PARITY}(x_1, x_2, \ldots, x_{F_{2t-1}}),\end{aligned}$$

which it then writes into the output cell. Since $F_{2t-1} = n$, this is the correct output.

In order to finish the description of the algorithm, we need to solve two problems: how to assign the processors to the cells for reading and how to choose the processors for writing to avoid a write conflict. There are $2^{(t+1)t/2}$ processors in each group named by binary strings of length $(t+1)t/2$. Let $s(0) = 0$ and define $s(k) = \sum_{l=1}^{k} l$ for $k > 0$. The memory cells and processors are assigned for reading and writing in such a way that after step k,

- in each group, the first k bits of the names of the active cells are identical and the remaining bits are arbitrary;
- in each group, the first $s(k-1)$ bits of the names of the active processors are identical and the remaining bits are arbitrary.

We show how the cells and processors are assigned for reading and writing during step $k+1$ to preserve these properties. During step $k+1$, each active processor of group j reads that memory cell of group $j+F_{2k-1}$ whose name agrees with the name of the processor from positions $s(k-1)+1$ through $s(k-1)+(t-1)$. After step k, the names of the active memory cells of group $j+F_{2k-1}$ have identical prefixes of length k. Hence after the READ operation, the names of active processors in group j have the same prefix π_{k+1} of length $s(k-1)+k=s(k)$; the remaining bits of their names may be arbitrary.

Now we must select processors for the WRITE operation. The processors of group j have to write into cells of group $j-F_{2k}$. For a cell with a name $u_1u_2\cdots u_{t-1}$, we select the processor with the name $\pi_{k+1}u_1u_2\cdots u_{t-1}00\cdots 0$ (this is the unique active processor that has a name with suffix $u_1u_2\cdots u_{t-1}00\cdots 0$). This processor sends the symbol $*$ into the chosen cell if and only if $u_{k+1}\neq \mathrm{PARITY}(x_j,x_{j+1},\ldots,x_{j+F_{2k+1}-1})$. Note that after this WRITE operation, the names of the active cells of group $j-F_{2k}$ have the same bit $k+1$, so together all bits from 1 through $k+1$ are fixed.

After the last READ, for the names of active processors, a prefix of length $s(t)=(t+1)t/2$ is fixed, that is, a complete name. Thus exactly one active processor remains in each group. The active processor of the first group writes the result into the output cell. □

Remark 4.2. Simple but tedious modifications in the above algorithm make it possible to remove the assumption that n is a Fibonacci number from Theorem 4.1.

Remark 4.3. Let $\Phi(y)=(\varphi(y)+2)\cdot(\varphi(y)+1)/2$ (hence $p=p(n):=n\cdot 2^{\Phi(n)}$ in Theorem 4.1). Knowing that $\log_b x\le\varphi(x)\le\log_b x+1.34$ for $b=\frac{1}{2}(3+\sqrt{5})$ (equation (1.1)), one can easily derive that $\Phi(n)\ge 0.25\log^2 n$ for all n and that $\Phi(n)\approx 0.26\log^2 n$ for sufficiently large n. For $n\ge 16$, we already have $\Phi(n)\le 4\log^2 n$. Hence $n\cdot 2^{0.25\log^2 n}\le p(n)\le n\cdot 2^{4\log^2 n}$ for $n\ge 16$.

5. Fast formula and circuit evaluation by CREW PRAMs with a linear number of processors. In the previous section, we proved that the function PARITY_n can be computed in time $\varphi(n)+1$ with subexponentially many processors. In this section, we show that PARITY_n can also be computed with a linear number of processors, while the computation time can be kept close to $\varphi(n)$. In a second step, the proof of this result is varied to obtain a method for evaluating Boolean circuits of depth d in $\varphi(2^d)+o(d)=0.72...d+o(d)$ steps on CREW PRAMs with a linear number of processors.

THEOREM 5.1. *There is an n-processor CREW PRAM M with n cells of wordsize 1 that computes PARITY_n in $\varphi(n)+O(\sqrt{\log n})$ steps.*

Proof. We may assume that $n=2^d$ for some integer d. The computation of M on input $x_1,\ldots,x_n$ proceeds in stages $i=1,\ldots,m$, where m is determined below. The input for stage i are n_i binary values $y_1^i,y_2^i,\ldots,y_{n_i}^i$ stored in n_i fixed cells so that the invariant

$$\mathrm{PARITY}(x_1,\ldots,x_n)=\mathrm{PARITY}(y_1^i,y_2^i,\ldots,y_{n_i}^i)$$

is maintained. A preprocessing phase in which groups of eight processors each compute the PARITY of eight bits in four steps (in the obvious binary-tree fashion) allows us to assume that we can start with $n_1 = n/8 = 2^{d-3}$ values $y_1^1, \dots, y_{n_1}^1$. The output of stage i, $1 \le i < m$, is the input sequence $y_1^{i+1}, y_2^{i+1}, \dots, y_{n_{i+1}}^{i+1}$ for the next stage; the output of stage m is a single bit $y_1^{m+1} = \text{PARITY}(x_1, \dots, x_n)$, the desired result. We now describe stage i, $i \ge 1$. We split the values $y_1^i, y_2^i, \dots, y_{n_i}^i$ into disjoint groups $y_{js+1}^i, \dots, y_{(j+1)s}^i$ of equal size s and compute PARITY within each group by the optimal algorithm of Theorem 4.1, which takes $\varphi(s) + 1$ steps and needs $s \cdot (2^{\varphi(s)} + 1) \le s \cdot 2^{\Phi(s)}$ cells and $s \cdot 2^{\Phi(s)}$ processors for each group, where $\Phi(s) = (\varphi(s) + 2) \cdot (\varphi(s) + 1)/2$ (cf. Remark 4.3). The $n_{i+1} := n_i/s$ output bits of the groups are $y_1^{i+1}, y_2^{i+1}, \dots, y_{n_{i+1}}^{i+1}$. Since n processors are available for a total of n_i bits, we can use sn/n_i processors for each group. If we define $w_i := \log(n/n_i)$, i.e., $n_i = 2^{d-w_i}$, then $s \cdot 2^{w_i}$ processors are available for each group. Thus any s with $s \cdot 2^{\Phi(s)} \le s \cdot 2^{w_i}$, or $\Phi(s) \le w_i$, is suitable. Since it is convenient to have groups of size a power of 2, we let u_i be the maximal integer u such that $\Phi(2^u) \le w_i$, and we define the group size s_i for stage i by $s_i := 2^{u_i}$. (Note that $w_i \ge w_1 = 3$ since $n_i \le n_1 = n/8$ and $\Phi(2^1) = 3$ by inspection; hence $u_i \ge 1$ for $i \ge 1$.) The only exception to this rule occurs in the last stage. Let $m := \min\{i \mid 2^{u_i} \ge n_i\}$, i.e., m is the minimal i with $w_i + u_i \ge d$. In stage m, we form one group of size $s_i := n_i$ and use $n_i \cdot 2^{\Phi(2^{n_i})} \le 2^{d-w_i} \cdot 2^{w_i} = 2^d = n$ processors to compute PARITY of all s_i bits left.

The correctness of the algorithm is obvious, as is the fact that only n processors and memory cells are used. We analyze the computation time. Note first that the equalities $s_i = n_i/n_{i+1}$ for $1 \le i < m$ and $s_m = n_m$ imply $\prod_{i=1}^m s_i = n_1 < n$. Using the inequalities $\log_b(z) \le \varphi(z) \le \log_b(z) + 1.34$, valid for all z, by equation (1.1), we may estimate the total number T of steps as follows:

$$\begin{aligned} T = \sum_{i=1}^{m} (\varphi(s_i) + 1) \le \sum_{i=1}^{m} (\log_b(s_i) + 2.34) &= \log_b\left(\prod_{i=1}^{m} s_i\right) + 2.34m \\ &\le \log_b(n) + 2.34m \le \varphi(n) + 2.34m. \end{aligned}$$

In order to prove Theorem 5.1, it remains to estimate m. We may conclude from the equality $2^{d-w_{i+1}} = n_{i+1} = n_i/s_i = 2^{d-w_i-u_i}$ that $w_{i+1} = w_i + u_i$ for $1 \le i < m$. This implies $w_i = w_1 + \sum_{j=1}^{i-1} u_j$ for $1 \le i \le m$. The number m is characterized as the smallest i that satisfies $w_i + u_i \ge d$; thus $m = \min\{i \mid w_1 + \sum_{j=1}^{i} u_j \ge d\}$. Now the desired estimate $m = O(\sqrt{\log n})$ is given by the following lemma (choose parameters $A = 2$, $q = 0$, $\eta = 0$, and $c = 1$). □

For later use, we formulate and prove the technical lemma missing in the previous proof in a slightly more general way than is needed here.

LEMMA 5.2. *Let integers $A \ge 2$ and $q \ge 0$ and some constant $\eta \in \{0, 1\}$ be fixed, let $d \ge 1$ be an integer, and let $c \in [1, d]$ be arbitrary. If the integer sequence v_i, $i \ge 1$, is defined recursively by*

$$v_i = \max\left\{ v \;\middle|\; \eta \cdot v + c \cdot v^q \cdot \Phi(A^v) \le \eta + c \cdot \Phi(A) + \sum_{j=1}^{i-1} v_j \right\} \quad \textit{for } i \ge 1$$

and $m = m(c, d)$ is defined by $m = \min\{i \mid \eta + c \cdot \Phi(A) + \sum_{j=1}^{i} v_j \ge d\}$, then

$$m = O((d^{q+1} \cdot c)^{1/(q+2)}).$$

(*The constant factor in this bound depends on A and q.*)

Proof. Obviously, $v_1 = 1$ and v_i, $i \geq 1$, is a nondecreasing sequence. For $l \geq 1$, let i_l denote the largest i such that $v_i \leq l$ (the fact that $v_i \geq 1$ implies that i_l is a well-defined integer). Further, let $i_0 = 0$. Let $S_l = i_l - i_{l-1}$, the number of indices i with $v_i = l$. Obviously,

$$v_1 + \cdots + v_{i_l} = S_1 + 2S_2 + \cdots + lS_l \quad \text{for } l \geq 1.$$

Let

$$l_0 = \min\{l \mid \eta + c \cdot \Phi(A) + S_1 + 2S_2 + \cdots + lS_l \geq d\}.$$

It is immediate from our definitions that $m \leq \sum_{l=1}^{l_0} S_l$. The following two claims are sufficient to prove the lemma.

CLAIM 1. *$S_l = O(c \cdot l^q)$ for $l \geq 1$.*

CLAIM 2. *$l_0 = O((d/c)^{1/(q+2)})$.*

Namely, using the two claims, we have

$$m \leq \sum_{l=1}^{l_0} S_l = \sum_{l=1}^{l_0} O(c \cdot l^q) = O(c \cdot l_0^{q+1}) = O\left(c \cdot \left(\frac{d}{c}\right)^{(q+1)/(q+2)}\right) = O((d^{q+1} \cdot c)^{1/(q+2)})$$

as desired.

Proof of Claim 1. Fix $l \geq 1$ and assume that $S_l \geq 1$. (If $S_l = 0$, there is nothing to show.) From our definitions, we have that

$$\eta \cdot l + c \cdot l^q \cdot \Phi(A^l) \leq \eta + c \cdot \Phi(A) + S_1 + 2S_2 + \cdots + (l-1)S_{l-1}$$

and

$$\eta + c \cdot \Phi(A) + S_1 + 2S_2 + \cdots + (l-1)S_{l-1} + l(S_l - 1) < \eta \cdot (l+1) + c \cdot (l+1)^q \cdot \Phi(A^{l+1}).$$

Adding these inequalities yields $l(S_l - 1) < \eta + c \cdot \left((l+1)^q \cdot \Phi(A^{l+1}) - l^q \cdot \Phi(A^l)\right)$, or

$$S_l \leq 1 + \frac{\eta}{l} + c \cdot \frac{1}{l} \cdot \left((l+1)^q \cdot \Phi(A^{l+1}) - l^q \cdot \Phi(A^l)\right).$$

Since $1 + \eta/l \leq 2$, to prove Claim 1, it suffices to show that

$$(l+1)^q \cdot \Phi(A^{l+1}) - l^q \cdot \Phi(A^l) = O(l^{q+1}). \tag{5.1}$$

Recall that for $u \geq 1$, we have $\Phi(A^u) = (\varphi(A^u)+1)(\varphi(A^u)+2)/2$ and (cf. Remark 1.1)

$$\beta u = \log_b A^u \leq \varphi(A^u) \leq \log_b A^u + 1.34 = \beta u + 1.34,$$

where $\beta = \log_b A = \log(A)/\log((3+\sqrt{5})/2)$. Thus we may estimate

$$\begin{aligned}
&2 \cdot \left((l+1)^q \cdot \Phi(A^{l+1}) - l^q \cdot \Phi(A^l)\right) \\
&\leq (l+1)^q(\beta(l+1) + 2.34)(\beta(l+1) + 3.34) - l^q(\beta l + 1)(\beta l + 2) \\
&= \beta^2(l+1)^{q+2} - \beta^2 l^{q+2} + O(l^{q+1}) \\
&= O(l^{q+1}).
\end{aligned}$$

This proves (5.1) and Claim 1.

Proof of Claim 2. The definition of l_0, respectively, $l_0 - 1$, implies that

$$\eta \cdot l_0 + c \cdot {l_0}^q \cdot \Phi(A^{l_0}) \leq \eta + c \cdot \Phi(A) + \sum_{j=1}^{i_{l_0-1}} v_j < d.$$

Substituting the inequality $\Phi(A^{l_0}) > \varphi(A^{l_0})^2/2 \geq \beta^2 l_0^2/2$ for $\beta = \log_b A$, we conclude that $c \cdot {l_0}^q \cdot \beta^2 l_0^2/2 < d$, which implies that $l_0 = O((d/c)^{1/(q+2)})$. Thus Claim 2 and Lemma 5.2 are proved. □

Computing PARITY$_n$ with $n = 2^d$ may be viewed as the problem of evaluating a certain Boolean formula that has the form of a complete binary tree of depth d with all operators equal to $\oplus$. A more general question is how fast Boolean formulas built from arbitrary binary operators may be evaluated or, even more generally, how fast Boolean circuits of depth d with gates of fan-in 2 may be evaluated. Of course, the best one can hope for is $\varphi(2^d) \approx 0.72d$ steps since, for example, OR$_n$ for $n = 2^d$ variables requires $\varphi(n)$ steps and has a circuit of depth d. In the following, we show that circuits of depth d can indeed be evaluated in $\varphi(2^d) + o(d)$ steps on CREW PRAMs with quite small hardware expenditure. Subsequently, we shall see that the special case of Boolean formulas (all gates have fan-out 1) allows a further reduction of the additive term $o(d)$.

For Boolean circuits, we use the standard notation as introduced, e.g., in [33, p. 9]. We assume that all gates have fan-in 1 or 2; there are no restrictions on the types of gates or on the fan-out. The depth of the circuit (i.e., the length of the longest path from an input to an output gate) is denoted by d and its size (i.e., the number of gates not counting the inputs) by s. In order to compute functions with several outputs, some of the gates are marked as output gates. Since our algorithms will determine the values at *all* gates, the positions of the output gates are irrelevant.

We will need to subdivide the gates of a circuit into *levels* of a certain size (the *width*) in the following (slightly unusual) sense. We say that a circuit C *can be arranged in* λ *levels of width up to* w if the set of gates of C can be partitioned into *levels* $L_1, \ldots, L_\lambda$ with $|L_l| \leq w$ for $1 \leq l \leq \lambda$ such that a wire may only connect an output of a gate on level L_i with an input of a gate on level L_j if $i < j$. The n inputs for C form a separate level L_0, which is not subject to the width condition.

Trivially, every circuit C of depth d and size s can be arranged in d levels of width up to s. Using a simple variant of Brent's scheduling principle [5], we obtain arrangements with smaller width.

LEMMA 5.3. *Let C be a Boolean circuit of size s and depth d, and let $w \geq 1$ be arbitrary. Then C can be arranged in $d + \lfloor s/w \rfloor$ levels of width up to w.*

Proof. As noted above, C can be arranged in d levels $L_1, \ldots, L_d$ of width up to s. For $1 \leq i \leq d$, level L_i is further subdivided into $\lceil |L_i|/w \rceil$ sublevels: $\lfloor |L_i|/w \rfloor$ of these consist of w gates and at most one consists of fewer than w gates. An order for the sublevels within one level is fixed arbitrarily. Clearly, there are overall at most $\lfloor s/w \rfloor$ sublevels with exactly w gates and at most d sublevels with fewer than w gates. □

The following technical lemma is the basis for our fast evaluation results for circuits and formulas.

LEMMA 5.4. *Let C be a Boolean circuit of size s that can be arranged in λ levels of width up to w. Let $\nu \geq 1$ be arbitrary. Then there is a CREW PRAM with $w \cdot 2^{2^\nu + \nu}$ processors and $s + w \cdot 2^{2^\nu + \nu}$ memory cells of wordsize 1 that for arbitrary inputs for*

C computes the values of all s gates of C in

$$\log_b(2^\lambda) + O\left(\frac{\lambda}{\nu}\right) = 0.72... \cdot \lambda + O\left(\frac{\lambda}{\nu}\right)$$

steps.

Proof. The computation proceeds in stages $t = 1, 2, \ldots, \lceil \lambda/\nu \rceil$. For simplicity, we assume that ν divides λ; to cover the general case, only slight changes are necessary. During stage t, the gates in levels L_l for $(t-1)\nu + 1 \le l \le t\nu$ are evaluated simultaneously by groups of processors working independently. The values of the gates are stored permanently in s designated cells. Consider one gate g at level L_l. By the fan-in restriction and the fact that wires run only from lower-numbered to higher-numbered levels, we know that the value of g is a function of the values of at most $2^{l-(t-1)\nu}$ gates in levels L_0 (the inputs), $L_1, \ldots, L_{l-(t-1)\nu}$, which are available at the beginning of stage t. By Fact 1.2, the value of g can be obtained in $\varphi(2^{l-(t-1)\nu}) + 1 \le \varphi(2^\nu) + 1$ steps by

$$2^{l-(t-1)\nu + 2^{l-(t-1)\nu} - 1} \le 2^{2^\nu + l - (t-1)\nu - 1}$$

processors using the same number of memory cells. Summing over the (up to w) gates on level L_l and summing over $(t-1)\nu + 1 \le l \le t\nu$, we see that stage t is finished in $\varphi(2^\nu) + 1$ steps and requires

$$w \cdot \sum_{r=1}^{\nu} 2^{2^\nu + r - 1} < w \cdot 2^{2^\nu + \nu}$$

processors and memory cells besides the memory cells for (permanently) storing the newly computed values of the gates in L_l, $(t-1)\nu + 1 \le l \le t\nu$. Since the same processors and memory cells can be used in all stages, the claimed bounds for these resources are proved. It remains to estimate the running time. Using equation (1.1), we see that the total number of steps made in all stages is bounded by

$$\frac{\lambda}{\nu} \cdot (\varphi(2^\nu) + 1) \le \frac{\lambda}{\nu} \cdot (\log_b(2^\nu) + 2.34) = \log_b(2^\lambda) + O\left(\frac{\lambda}{\nu}\right). \qquad \Box$$

It is now only a matter of adjusting the parameters w and ν to obtain a CREW PRAM with fewer than s processors that can evaluate a circuit C of size s and depth d in time $0.72... \cdot d + o(d)$.

THEOREM 5.5. *Let $f : \{0,1\}^n \to \{0,1\}^m$ be a Boolean function that is computed by a circuit C of depth d and size s.*

(a) *Let $\nu \ge 1$ and p be arbitrary. Then f can be computed by a CREW PRAM with p processors and $p + s$ memory cells of wordsize* 1 *in* $\varphi(2^d) + O\left(2^{2^\nu + \nu} \cdot s/p\right) + O\left(d/\nu\right)$ *steps.*

(b) *If p is such that* $\log\log(pd/s) \ge 3$, *then there is a CREW PRAM with p processors and $p + s$ memory cells of wordsize* 1 *that computes f in* $\varphi(2^d) + O(d/\log\log(pd/s))$ *steps.*

(c) *Assume that* $\log\log(pd/s) \ge 3$. *If $p \ge s/\sqrt{d}$, the running time in* (b) *is* $\varphi(2^d) + O(d/\log\log d)$; *if $p \ge s/(d/\log d)$, it is* $\varphi(2^d) + O(d/\log\log\log d)$.

Proof. (a) If $2^{2^\nu + \nu} > p/2$, there is nothing to show since C can then be evaluated in $O(s) = O(p \cdot (s/p)) = O(2^{2^\nu + \nu} \cdot (s/p))$ steps by one processor. Thus assume that $p \ge 2^{2^\nu + \nu + 1}$. Let $w = \lfloor p/2^{2^\nu + \nu} \rfloor \ge 2$, and apply Lemma 5.3 to see that C can be arranged in $\lambda = d + \lfloor s/w \rfloor$ levels of width w. By Lemma 5.4, C can be evaluated by $w \cdot 2^{2^\nu + \nu} \le p$ processors in time

$$\log_b(2^\lambda) + O\left(\frac{\lambda}{\nu}\right) = \log_b(2^d) + O\left(\frac{s}{w}\right) + O\left(\frac{d}{\nu}\right) + O\left(\frac{s}{w \cdot \nu}\right)$$
$$= \log_b(2^d) + O\left(\frac{d}{\nu}\right) + O\left(\frac{s}{w}\right).$$

Now it suffices to observe that

$$\frac{s}{w} = \frac{s}{\lfloor p/2^{2^\nu+\nu} \rfloor} \le \frac{2s}{p/2^{2^\nu+\nu}} = 2 \cdot \frac{s}{p} \cdot 2^{2^\nu+\nu}.$$

(b) We use (a) with $\nu := \lfloor \log\log(pd/s) - 2 \rfloor \ge 1$. Then $d/\nu = O\left(d/(\log\log(pd/s))\right)$ and

$$2^{2^\nu+\nu} \cdot \frac{s}{p} \le 2^{2^{\nu+1}} \cdot \frac{s}{p} \le 2^{\frac{1}{2}\log(pd/s)} \cdot \frac{s}{p} = O\left(\frac{d}{\sqrt{pd/s}}\right) = O\left(\frac{d}{\log\log(pd/s)}\right).$$

(c) This follows immediately from (b). □

For the special case of formulas, i.e., circuits in which all gates have fan-out 1, a better additive error term can be achieved, as noted in the following.

THEOREM 5.6. *Let $n = 2^d$. If the Boolean function $f : \{0,1\}^n \to \{0,1\}$ can be represented by a formula F of depth d, then f can be evaluated by a CREW PRAM with p processors and $p + n$ memory cells in*

$$\varphi(n) + \left\lceil \frac{n}{p} \right\rceil + O\left(\frac{\log n}{\log\log n}\right)$$

steps.

Proof. We may assume without loss of generality that $p \le n$ and that p is a power of 2. Assume first that $p = n$. In a first phase, the $2^{d - \lceil \sqrt{d} \rceil}$ subformulas of F are evaluated that correspond to the $2^{d-\lceil \sqrt{d} \rceil}$ subtrees of depth $\lceil \sqrt{d} \rceil$ at the bottom of F. Clearly, each such subformula can be evaluated in $\lceil \sqrt{d} \rceil + 1$ steps by $2^{\lceil \sqrt{d} \rceil}$ processors in the same number of memory cells of wordsize 1, even in an EREW fashion. Overall, n processors and cells are sufficient. We are left with the problem of evaluating a formula of depth $d' = d - \lceil \sqrt{d} \rceil$. Trivially, this formula is a circuit that can be arranged in levels of width at most $w := 2^{d'}$. By Lemma 5.4, for arbitrary ν, this circuit can be evaluated by $p = w \cdot 2^{2^\nu + \nu}$ processors and $n + w \cdot 2^{2^\nu+\nu}$ memory cells in $\log_b(2^{d'}) + O(d'/\nu)$ steps. We choose $\nu = \lfloor \frac{1}{3} \log\log n \rfloor = \lfloor \frac{1}{3} \log d \rfloor$ and obtain that $w \cdot 2^{2^\nu+\nu} \le 2^{d - \lceil \sqrt{d} \rceil} \cdot 2^{d^{1/3} + (1/3)\log d} \le 2^d = n$ (for d sufficiently large) and, of course, $\log_b(d') \le \varphi(2^d)$ and $O(d'/\nu) = O(d/\log d)$. In the case where $p < n$, we first reduce the size of the formula to p leaves by having each of the p subformulas of F of depth $d - \log p$ evaluated by one processor in n/p steps. Afterwards, we proceed as before. □

6. Many-valued formulas and circuits. Some of the previous results can be extended to k-valued formulas and circuits, i.e., devices that compute functions $\{0, \ldots, k-1\}^n \to \{0, \ldots, k-1\}^m$ for some $k \ge 3$ and are built from gates that compute functions

$$\otimes : \{0, \ldots, k-1\}^2 \to \{0, \ldots, k-1\}.$$

For CREW PRAMs that compute such functions, the n arguments are stored in input memory cells of wordsize $\|k-1\|$, each argument in a separate cell. A component of the result is either given as a single value in an output cell of wordsize $\|k-1\|$ or coded in binary in $\|k-1\|$ cells of wordsize 1. We start with considering *simple* k-valued formulas, that is, functions $F(x_1,\ldots,x_n) = x_1 \otimes x_2 \otimes \cdots \otimes x_n$ for an associative operator $\otimes$ over $\{0,\ldots,k-1\}$ with values in $\{0,\ldots,k-1\}$. Such functions are direct generalizations of the functions AND_n, OR_n, and PARITY_n from the case where $k=2$. Theorem 4.1 may be generalized as follows.

THEOREM 6.1. *A simple k-valued formula F of size n can be evaluated by a CREW PRAM M with $p = n \cdot k^{(t+1)t/2}$ processors and $n \cdot (k^{t-1}+1)$ memory cells in $t = \varphi(n)+1$ steps, where each of the common memory cells of M, except the cells used for input and output, are of wordsize* 1.

The proof of this theorem is almost identical to that of Theorem 4.1. The only difference is that instead of binary strings, we use strings over the alphabet $\{0,\ldots,k-1\}$ as names for cells and processors. During the ith WRITE, if a group of processors wants to send a number $s \in \{0,\ldots,k-1\}$ to a group of cells $\mathcal{C}$, the symbol $*$ is written to all cells in $\mathcal{C}$ with a name whose ith position differs from s, thus changing the state of these cells to "dead."

Remark 6.2. We note that the result of such a computation can be written in binary in $\|k-1\|$ memory cells in one extra step. Indeed, we can prepare k different cells, all storing the same symbol. The processor that knows $F(x)$ after the READ of step $\varphi(n)+1$ marks one of these cells, namely the jth cell, where $j = F(x)+1$. During step $\varphi(n)+2$, another $k \cdot \|k-1\|$ processors read these cells in parallel ($\|k-1\|$ processors for each cell); during the following WRITE, the $\|k-1\|$ processors that encountered the marked cell in parallel write the binary code of $F(x)$. In this way, the algorithm uses only memory cells of wordsize 1 (except for the input cells).

Remark 6.3. It is interesting to note that, in many cases, a Boolean formula that is not nicely balanced like the formulas of Theorem 5.6 can be considered as the restriction of a simple formula over a k-valued domain. (The well-known fact that any unbalanced Boolean formula of, say, size n can be restructured to get an equivalent one of depth $O(\log n)$ does not help in constructing a fast CREW algorithm for the formula because the constant factor in front of the logarithm in the depth bound (a factor of about 3) outweighs the saving from $\log n$ down to $\varphi(n)$.) As an example, consider the Horner-type formula

$$F(x_1,\ldots,x_n) = (\cdots((((x_1 \wedge x_2) \vee x_3) \wedge x_4) \vee x_5) \cdots \wedge x_n)$$

of size $n = 2m$ and depth $n-1$, which is extremely unbalanced. We define an associative operator $\otimes$ on the 3-valued domain $\{k,p,g\}$ by $u \otimes k = k$, $u \otimes g = g$, and $u \otimes p = u$ for $u \in \{k,p,g\}$. Furthermore, we define the functions $G : \{0,1\}^2 \to \{k,p,g\}$ and $H : \{0,1\} \times \{k,p,g\} \to \{0,1\}$ by $G(x,1) = g$, $G(1,0) = p$, $G(0,0) = k$, $H(x,k) = 0$, $H(x,p) = x$, and $H(x,g) = 1$ for $x \in \{0,1\}$. Then it is easy to see that

$$F(x_1,\ldots,x_n) = H(x_1, G(x_2,x_3) \otimes G(x_4,x_5) \otimes \cdots \otimes G(x_{n-2},x_{n-1}) \otimes G(x_n,0)).$$

Therefore, by Theorem 6.1, one can evaluate $F(x_1,\ldots,x_n)$ in $\varphi(n)+4$ steps on a CREW PRAM with subexponentially many processors and cells. The degree of F is n; hence at least $\varphi(n)$ steps are necessary to compute F by Fact 1.4.

We generalize Theorem 5.1 to k-valued domains.

THEOREM 6.4. *Let $F(x_1,\ldots,x_n) = x_1 \otimes \cdots \otimes x_n$ be a simple k-valued formula. Then there is an n-processor CREW PRAM M with n memory cells of wordsize $\|k-1\|$ that evaluates F in $\varphi(n) + O(\sqrt{\log n \cdot \log k})$ steps.*

Proof. We may assume that $k \leq n$ since $\log n + 1$ steps are certainly sufficient, which is $O(\sqrt{\log n \cdot \log k})$ for $k > n$. We proceed essentially as in the proof of Theorem 5.1. The input for stage i consists of $n_i = 2^{d-w_i}$ values $y_1^i, \ldots, y_{n_i}^i$ in $\{0, \ldots, k-1\}$ so that $F(x_1, \ldots, x_n) = y_1^i \otimes \cdots \otimes y_{n_i}^i$. In stage i, the y_j^i's, $1 \leq j \leq n_i$, are subdivided into $n_{i+1} := n_i/s_i$ blocks, or groups, of $s_i = 2^{u_i}$ consecutive values; the operation $\otimes$ is applied within each group to yield one new value y_j^{i+1}. This computation takes time $\varphi(s_i) + 1$ according to Theorem 6.1. The maximal group size that we can afford without exceeding the processor bound can be easily calculated. Let u_i be the largest u such that $\lceil \log k \rceil \cdot \Phi(2^u) \leq w_i$. Then for each group, $s_i \cdot 2^{\lceil \log k \rceil \cdot \Phi(2^{u_i})} \geq s_i \cdot k^{\Phi(s_i)}$ processors are available, which is sufficient for applying Theorem 6.1. For this to work, we need $u_i \geq 1$, or $\lceil \log k \rceil \cdot \Phi(2^1) \leq w_i$. In order to reach such a situation, we start with a preprocessing phase of $3\lceil \log k \rceil = O(\sqrt{\log n \log k})$ steps in each of which simply adjacent pairs of values y_{2l-1}^i and y_{2l}^i are combined to form y_l^{i+1}. Thus the first stage starts with $n_1 = 2^{d-w_1}$ values $y_1^1, \ldots, y_{n_1}^1$ with $3\lceil \log k \rceil \leq w_1$. Stage i is the last stage if $n_i \leq 2^{u_i}$; in this case, there is only one group of size $s_i = n_i$. The number of stages is denoted by m. The analysis now is very similar to the proof of Theorem 5.1. Let T be the computation time (without preprocessing). Then $T \leq \varphi(n_1) + 2.34m \leq \varphi(n) + 2.34m$. To estimate m, we prove just as in Theorem 5.1 that

$$u_i = \max\left\{ u \;\middle|\; \lceil \log k \rceil \cdot \Phi(2^u) \leq \sum_{j=1}^{i-1} u_j + 3\lceil \log k \rceil \right\} \quad \text{for } 1 \leq i \leq m,$$

and

$$m = \min\left\{ i \;\middle|\; 3\lceil \log k \rceil + \sum_{j=1}^{i} u_j \geq d \right\}.$$

Lemma 5.2, applied with parameters $A = 2$, $q = 0$, $\eta = 0$, and $c = \lceil \log k \rceil$, yields $m = O(\sqrt{d \log k}) = O(\sqrt{\log n \log k})$, as desired. □

The last algorithm can be modified so that it uses only memory cells of wordsize 1. The simple idea is to store the intermediate results $y_1^i, \ldots, y_{n_i}^i$ passed from stage $i-1$ to stage i in binary encoding. For (sequentially) reading and writing these codes, in each stage $2\|k-1\| = O(\log k)$ extra steps are sufficient. In this way, the computation time increases to $\varphi(n) + O\big(\sqrt{\log n \cdot \log^3 k}\big)$.

Finally, we generalize Theorems 5.5 and 5.6 to k-valued domains. The notion of a circuit consisting of gates with fan-in 2 that compute functions $\{0, \ldots, k-1\}^2 \to \{0, \ldots, k-1\}$ is an easy generalization of the binary case. (The prime example here is an arithmetic circuit over a ring or a field with k elements.) The definitions of the depth of such a circuit and of arranging the circuit in λ levels of width up to w (cf. §5) carry over directly, as does Lemma 5.3.

THEOREM 6.5. *Let C be a circuit of depth d consisting of s gates over the domain $\{0, \ldots, k-1\}$.*

(a) *Let $\nu \geq 1$ and p be arbitrary. Then C can be evaluated by a CREW PRAM with p processors and $s + p$ memory cells of wordsize $\|k-1\|$ in*

$$\varphi(2^d) + O\left(2^{\nu+1} \cdot k^{2^\nu} \cdot \frac{s}{p}\right) + O\left(\frac{d}{\nu}\right)$$

steps.

(b) *If p satisfies $\log\log(pd/s) - \log\log k \geq 3$, then there is a CREW PRAM with p processors and memory cells of wordsize $\|k-1\|$ that evaluates C in*

$$\varphi(2^d) + O\left(\frac{d}{\log\log(pd/s) - \log\log k}\right)$$

steps. (*Note that this is $\varphi(2^d) + o(d)$ whenever k is fixed and $p = \omega(s/d)$.*)

Proof. We may assume that $2^{\nu+1} \cdot k^{2^\nu} \leq p/2$ since otherwise even one processor can evaluate C in $2s = O\left(2^{\nu+1} \cdot k^{2^\nu} \cdot s/p\right)$ steps. Let $w := \lfloor p/(2^{\nu+1} \cdot k^{2^\nu})\rfloor \geq 2$, and arrange C in $\lambda = d + \lfloor s/w \rfloor$ levels of width w (cf. Lemma 5.3). As in the proof of Lemma 5.4, the s gates are evaluated in stages $t = 1, 2, \ldots, \lambda/\nu$. Evaluating gate g at level l, $(t-1)\nu + 1 \leq l \leq t\nu$, with the values of all the gates in levels $L_0, L_1, \ldots, L_{l-(t-1)\nu}$ already available amounts to computing the value of a function with $2^{l-(t-1)\nu}$ inputs from $\{0, \ldots, k-1\}$ and an output in $\{0, \ldots, k-1\}$. This can be done in $\varphi(2^{l-(t-1)\nu}) + 1 \leq \varphi(2^\nu) + 1$ steps by $k^{2^{l-(t-1)\nu}} \cdot 2^{l-(t-1)\nu}$ processors. (The method is practically the same as in the binary case. For each of the at most $k^{2^{l-(t-1)\nu}}$ possible inputs, a team of $2^{l-(t-1)\nu}$ processors checks whether the actual input equals the input associated with the team. This is done by computing the OR of $2^{l-(t-1)\nu}$ bits. The unique successful team writes the result into the output cell.) The total number of processors used is bounded by $w \cdot 2 \cdot 2^\nu \cdot k^{2^\nu} = w \cdot 2^{\nu+1} \cdot k^{2^\nu} \leq p$. Proceeding exactly as in the proofs of Lemma 5.4 and Theorem 5.5, we may estimate the running time by $\varphi(2^d) + O\left(2^{\nu+1} \cdot k^{2^\nu} \cdot s/p\right) + O\left(d/\nu\right)$.

(b) Define $\nu := \lfloor \log\log(pd/s) - \log\log k - 2 \rfloor \geq 1$. Clearly, d/ν is bounded as required. Moreover,

$$\begin{aligned} 2^{\nu+1} \cdot k^{2^\nu} \cdot \frac{s}{p} < k^{2^{\nu+1}} \cdot \frac{s}{p} &\leq k^{2^{(\log\log(pd/s)-1)-\log\log k}} \cdot \frac{s}{p} \\ &= k^{(1/2)\cdot\log(pd/s)/\log k} \cdot \frac{s}{p} = \frac{d}{\sqrt{pd/s}} \\ &= O\left(\frac{d}{\log\log(pd/s) - \log\log k}\right). \end{aligned}$$

Thus both O terms in (a) can be bounded as claimed. □

In the case of k-valued formulas of depth d and size $n = 2^d$ (e.g., balanced arithmetic expressions over finite fields), the last result may be somewhat sharpened, in analogy to Theorem 5.6.

THEOREM 6.6. *Let $n = 2^d$ and let $k \leq 2^{\sqrt{d}/4}$. If the function $f : \{0, \ldots, k-1\}^n \to \{0, \ldots, k-1\}$ can be represented by a formula F of depth d (with arbitrary binary operators over $\{0, \ldots, k-1\}$), then f can be evaluated by a CREW PRAM with $p \leq n$ processors and $p + n$ memory cells of wordsize $\|k-1\|$ in*

$$\varphi(n) + \left\lceil \frac{n}{p} \right\rceil + O\left(\frac{\log n}{\log\log n - 2\log\log k}\right)$$

steps. (*If $\log k = o(\sqrt{\log n})$ and $p = \omega(n/\log n)$, the number of steps is bounded by $\varphi(n) + o(\log n)$.*)

Proof. We follow the proof of Theorem 5.6 and consider here only the case $p = n$. Subformulas of depth $\lceil \sqrt{d} \rceil$ may be evaluated in $\lceil \sqrt{d} \rceil + 1$ steps in the k-valued case as well. The remaining formula F' can be arranged in $d' = d - \lceil \sqrt{d} \rceil$ levels of width at most $w := 2^{d'}$. Let $\nu := \lfloor \frac{1}{2}\log d - \log\log k - 1 \rfloor$. Then $\nu \geq 1$ by the assumption

$k \leq 2^{\sqrt{d}/4}$. By the proof of Theorem 6.5, $w \cdot 2^{\nu+1} \cdot k^{2^\nu}$ processors can evaluate F' in $\varphi(2^{d'}) + O(d'/\nu) = \varphi(n) + O(d/(\log\log n - 2\log\log k))$ steps. It remains to note that $w \cdot 2^{\nu+1} \cdot k^{2^\nu} \leq 2^{d-\lceil\sqrt{d}\rceil} \cdot k^{2^{\nu+1}} \leq 2^{d-\lceil\sqrt{d}\rceil} \cdot k^{\sqrt{d}/\log k} = 2^d = n$. □

The algorithms described in Theorems 6.4, 6.5, and 6.6 are unsatisfying in that they lead to computation times of $\varphi(n) + o(\log n)$ only if k is sufficiently small relative to n. Designing feasible algorithms that run in almost optimal time for arbitrary k seems to be difficult. However, in §8, we present a feasible algorithm for computing an important $(n+1)$-valued function, namely the sum of n bits, in almost optimal time.

7. Parallel prefix and addition. Let $\otimes$ be an associative operator over some domain. We say that a PRAM computes the parallel prefix for the product $x_1 \otimes x_2 \otimes \cdots \otimes x_n$ if on input $x_1, x_2, \ldots, x_n$, the PRAM computation results in the values of the n products

$$x_1,\ x_1 \otimes x_2,\ x_1 \otimes x_2 \otimes x_3, \ldots,\ x_1 \otimes \cdots \otimes x_n$$

being stored in n fixed memory cells. Parallel-prefix computation is a fundamental problem which has been studied extensively for different computational models. Optimal realizations for the circuit model can be found in [19] and [13]. For unbounded fan-in circuits, which relate to the CRCW PRAM model, see for example [10].

In this section, we show that a parallel prefix for k-valued domains can be computed on a CREW PRAM within practically the same complexity bounds as the single product $x_1 \otimes x_2 \otimes \cdots \otimes x_n$ (Theorem 6.4).

Remark 7.1. In the construction of an adder for two n-bit numbers by Ladner and Fischer [19], a k-valued circuit for the parallel-prefix problem is described that has depth $\log n$ and size $4n$. We could apply the general Theorem 6.5 to obtain a CREW PRAM algorithm for parallel prefix with n processors that runs in time $\varphi(n) + o(n)$ if $\log\log\log n - \log\log k = \omega(1)$, which means that $k = o(\log\log n)$. The direct construction presented in the following works for k with $\log k = o(\log n)$.

We start with a simple lemma.

LEMMA 7.2. *Let $\otimes$ be an associative operator $\{0, \ldots, k-1\}^2 \to \{0, \ldots, k-1\}$ over a k-valued domain. Then there is an EREW PRAM M with n processors and n memory cells of wordsize $\|k-1\|$ that computes the parallel prefix for $x_1 \otimes x_2 \otimes \cdots \otimes x_n$ in $\lceil \log n \rceil + 1$ steps.*

Proof. Without loss of generality, we assume that $n = 2^d$. During the computation, the arguments $x_1, \ldots, x_n$ are divided into blocks; after step t, these blocks consist of 2^{t-1} variables. Namely, for each $i \leq n$, let $B_i^1 = \{x_i\}$, and for $t > 1$, $B_i^{t+1} = B_{2i-1}^t \cup B_{2i}^t$. Thus $B_i^t = \{x_{(i-1)\cdot 2^{t-1}+1}, x_{(i-1)\cdot 2^{t-1}+2}, \ldots, x_{i\cdot 2^{t-1}}\}$. Let $\prod B_i^t$ denote the product of the elements of B_i^t, that is, $x_{(i-1)\cdot 2^{t-1}+1} \otimes x_{(i-1)\cdot 2^{t-1}+2} \otimes \cdots \otimes x_{i\cdot 2^{t-1}}$. By $\mathrm{prefix}(s, B_i^t)$ for $x_s \in B_i^t$, we denote $x_{(i-1)\cdot 2^{t-1}+1} \otimes x_{(i-1)\cdot 2^{t-1}+2} \otimes \cdots \otimes x_s$. The algorithm maintains the following invariant:

- *After step t, for each $s \leq n$, processor P_s knows $\prod B_i^t$ and* $\mathrm{prefix}(s, B_i^t)$, *where B_i^t is the block that contains x_s. Moreover, cell C_s stores $\prod B_i^t$.*

The first step is simple: each processor P_s for $s \leq n$ reads from cell C_s. To describe step $t+1$, assume that the property above holds up to step t. Let $s \leq n$ and $x_s \in B_i^{t+1} = B_{2i-1}^t \cup B_{2i}^t$. If $x_s \in B_{2i-1}^t$, then during step $t+1$, processor P_s reads from cell $C_{s+2^{t-1}}$, which stores $\prod B_{2i}^t$ (since $x_{s+2^{t-1}} \in B_{2i}^t$). If $x_s \in B_{2i}^t$, then P_s reads from cell $C_{s-2^{t-1}}$, which stores $\prod B_{2i-1}^t$. It is easy to check that this does not lead to a read conflict. Note that in both cases, after the READ operation, processor P_s knows

$\prod B_{2i}^t$ as well as $\prod B_{2i-1}^t$. Then P_s computes $\prod B_i^{t+1}$ and writes it into cell C_s. Also, P_s computes prefix(s, B_i^{t+1}), knowing that prefix$(s, B_i^{t+1}) =$ prefix(s, B_{2i-1}^t) if $x_s \in B_{2i-1}^t$ and prefix$(s, B_i^{t+1}) = \prod B_{2i-1}^t \otimes$ prefix(s, B_{2i}^t) if $x_s \in B_{2i}^t$. The computation stops when $B_1^t = \{x_1, \ldots, x_n\}$; hence $t = \log n + 1$. During the WRITE phase of step t, each processor P_s writes prefix(s, B_1^t) into C_s instead of $\prod B_1^t$. Therefore, after step t, we get the correct output. □

THEOREM 7.3. *Let $\otimes$ be an associative operator over a k-valued domain. The parallel prefix for $x_1 \otimes x_2 \otimes \cdots \otimes x_n$ can be computed by an n-processor CREW PRAM in $\varphi(n) + O(\sqrt{\log k \cdot \log n})$ steps using $2n$ memory cells of wordsize $\|k-1\|$.*

Proof. We may assume that $k \leq n$ and $n = 2^d$. (If $k > n$, use the algorithm of Lemma 7.2. If n is not a power of 2, take $n' = 2^{\lfloor \log n \rfloor}$ and compute the parallel prefix for $y_1 \otimes y_2 \otimes \cdots \otimes y_{n'}$, where $y_i = x_{2i-1} \otimes x_{2i}$ for $i \leq n - n'$ and $y_i = x_{i+(n-n')}$ for $n - n' < i \leq n'$. Then the products $x_1 \otimes \cdots \otimes x_j$, $1 \leq j \leq n$, can be obtained in two additional steps.) The idea of the algorithm is to divide the input variables into disjoint blocks and to compute the parallel prefix of each block. Then the input variables are partitioned into larger blocks and the parallel prefix of each block is computed using the previous results. The second step is repeated until there is only one block consisting of all variables. At each time during the computation, cell C_r for $r \leq n$ stores the value of the product $x_l \otimes x_{l+1} \otimes \cdots \otimes x_r$, where x_l is the beginning of the block that currently contains x_r. The remaining n memory cells are used for auxiliary data. After the last step, since there is a single block containing all variables, cell C_r stores the value of $x_1 \otimes x_2 \otimes \cdots \otimes x_r$, which is the correct result.

The computation consists of a preprocessing phase and stages $i = 1, \ldots, m$. At the beginning of stage i, the following situation is given. The input variables are divided into blocks $B_{i,1}, B_{i,2}, \ldots, B_{i,n_i}$ of the same size—say 2^{w_i}; hence $2^{w_i} \cdot n_i = n$, or $n_i = 2^{d-w_i}$. For each block $B_{i,p} = \{x_{j_{i,p}}, x_{j_{i,p}+1}, \ldots, x_{j_{i,p+1}-1}\}$, there is a memory cell that stores

$$\prod B_{i,p} = x_{j_{i,p}} \otimes x_{j_{i,p}+1} \otimes \cdots \otimes x_{j_{i,p+1}-1}.$$

Let $y_p = \prod B_{i,p}$, for $p \leq n_i$. Further, cell C_r stores $x_{j_{i,p}} \otimes x_{j_{i,p}+1} \otimes \cdots \otimes x_r$, where $B_{i,p}$ is the block that contains x_r. Stage i consists of the following computation: M splits $y_1, y_2, \ldots, y_{n_i}$ into blocks $Y_p = \{y_{(p-1)\cdot s_i+1}, y_{(p-1)\cdot s_i+2}, \ldots, y_{p\cdot s_i}\}$ of some size s_i, where $1 \leq p \leq n_{i+1} := n_i/s_i$, and computes the parallel prefix of the values $y_1, y_2, \ldots, y_{n_i}$ within each block Y_p (details are given below). Each block $B_{i+1,p}$ consists of s_i blocks of the form $B_{i,j}$, namely,

$$B_{i+1,p} = B_{i,(p-1)\cdot s_i+1} \cup B_{i,(p-1)\cdot s_i+2} \cup \cdots \cup B_{i,p\cdot s_i}.$$

Of course, there are n_{i+1} such blocks $B_{i+1,p}$. Assume that x_r is in $B_{i,q} \subseteq B_{i+1,p}$. At the beginning of stage $i+1$, cell C_r must store the value of $x_{j_{i+1,p}} \otimes x_{j_{i+1,p}+1} \otimes \cdots \otimes x_r$. Note that

$$\begin{aligned} &x_{j_{i+1,p}} \otimes x_{j_{i+1,p}+1} \otimes \cdots \otimes x_r \\ (7.1) \quad &= \left(\prod B_{i,(p-1)\cdot s_i+1} \otimes \prod B_{i,(p-1)\cdot s_i+2} \otimes \cdots \otimes \prod B_{i,q-1}\right) \\ &\qquad \otimes (x_{j_{i,q}} \otimes x_{j_{i,q}+1} \otimes \cdots \otimes x_r) \\ &= (y_{(p-1)\cdot s_i+1} \otimes y_{(p-1)\cdot s_i+2} \otimes \cdots \otimes y_{q-1}) \otimes (x_{j_{i,q}} \otimes x_{j_{i,q}+1} \otimes \cdots \otimes x_r). \end{aligned}$$

The value of $x_{j_{i,q}} \otimes x_{j_{i,q}+1} \otimes \cdots \otimes x_r$ is stored in cell C_r. The product $y_{(p-1)\cdot s_i+1} \otimes y_{(p-1)\cdot s_i+2} \otimes \cdots \otimes y_{q-1}$ is computed during stage i as one of the prefixes of block Y_p

and stored in some fixed memory cell. Some processor reads this cell as well as cell C_r, computes the product $x_{j_{i+1,p}} \otimes x_{j_{i+1,p}+1} \otimes \cdots \otimes x_r$ according to equation (7.1), and stores the result in C_r.

It remains to fix the sizes of the blocks and the algorithm used within the stage and to analyze the running time.

We start with a preprocessing phase in which the parallel prefix is computed within blocks of length 2^{w_1}, where $w_1 = 1 + 3\lceil \log k \rceil$, by the algorithm of Lemma 7.2. This takes $2 + 3\lceil \log k \rceil = O(\sqrt{\log n \log k})$ steps.

During stage i, $i \geq 1$, we must compute the parallel prefix within each block Y_p for $1 \leq p \leq n_{i+1}$. Each single prefix product $y_{(p-1)\cdot s_i+1} \otimes y_{(p-1)\cdot s_i+2} \otimes \cdots \otimes y_{q-1}$ is computed by a separate group of processors and memory cells by the algorithm of Theorem 6.1. For each such group, we employ $p(s_i) = s_i \cdot k^{\Phi(s_i)}$ processors (and fewer than $p(s_i)$ memory cells), where $\Phi(s_i) = (\varphi(s_i) + 2) \cdot (\varphi(s_i) + 1)/2$, as before. Since n_i products are to be computed, $n_i \cdot s_i \cdot k^{\Phi(s_i)}$ processors are required overall. If we let u_i be the largest integer u that satisfies

$$n_i \cdot 2^{u_i} \cdot 2^{\lceil \log k \rceil \cdot \Phi(2^{u_i})} \leq n,$$

which means that $u_i + \lceil \log k \rceil \cdot \Phi(2^{u_i}) \leq w_i$, then we may choose $s_i = 2^{u_i}$. By the preprocessing stage, we have $u_i \geq 1$ for all i. The last stage, m, is characterized as the minimal i with $n_i \leq 2^{u_i}$. Here we choose $s_i = n_i$. The analysis of the running time (disregarding the preprocessing phase) is now practically the same as in Theorem 6.4. Stage i takes $(\varphi(s_i) + 1) + 2 = \varphi(s_i) + 3 \leq \log_b(s_i) + 4.34$ steps, and the overall time T can be estimated by $T \leq \varphi(n) + 4.34m$. In order to estimate m, we note that u_i is given by the recursion

$$u_i = \max\left\{ u \;\middle|\; u + \lceil \log k \rceil \cdot \Phi(2^u) \leq 1 + 3\lceil \log k \rceil + \sum_{j=1}^{i-1} u_j \right\} \quad \text{for } 1 \leq i \leq m$$

and

$$m = \min\left\{ i \;\middle|\; 1 + 3\lceil \log k \rceil + \sum_{j=1}^{i} u_j \geq d \right\}.$$

In this situation, Lemma 5.2 is applicable (with $A = 2$, $\eta = 1$, $c = \lceil \log k \rceil$, and $q = 0$); it yields $m = O(\sqrt{c \cdot d}) = O(\sqrt{\log n \log k})$. □

A CREW PRAM M adding two n-bit numbers gets the input bits in separate cells and has to generate the binary representation of the sum of these numbers, each bit in a separate cell.

COROLLARY 7.4. *For each n, there is an $(n+1)$-processor CREW PRAM with $2(n+1)$ memory cells of wordsize 2 that adds two n-bit numbers in $\varphi(n) + O(\sqrt{\log n})$ steps.*

Proof. When computing in parallel a sum of two binary numbers, the main problem is to compute the carry bits. Once all carry bits are known, the sum can be computed in a few parallel steps. Let the carry propagation operator $\otimes : \{0, 1, p\}^2 \to \{0, 1, p\}$ be defined by

$$p \otimes x = x, \qquad 0 \otimes x = 0, \qquad 1 \otimes x = 1.$$

It is well known that this operator is associative and that the ith carry bit c_i equals $x_i \otimes x_{i-1} \otimes \cdots \otimes x_0$, where $x_0 = 0$ and, for $i > 0$, $x_i = 1$ if both added numbers have

1's in position i, $x_i = 0$ if there are two 0's in position i, and $x_i = p$ if there is a 0 and a 1 in position i. If we define $x \otimes' y = y \otimes x$, then $c_i = x_0 \otimes' x_1 \otimes' \cdots \otimes' x_i$. Therefore, to compute the carry bits, we have to compute the parallel prefix of $x_0 \otimes' x_1 \otimes' \cdots \otimes' x_n$. By Theorem 7.3, this can be done in $\varphi(n) + O(\sqrt{\log n})$ steps by a CREW PRAM with $n+1$ processors and $2(n+1)$ memory cells of wordsize 2. In a few additional steps, this machine computes each bit of the sum. □

Remark 7.5. The best currently known construction of a circuit for adding two n-bit numbers was given by Krapchenko (see [33, p. 42]). This circuit has depth $\log n + O(\sqrt{\log n})$ and size $O(n)$. Applying Theorem 5.5 to this circuit yields a CREW PRAM algorithm with running time $\varphi(n) + O(\log n / \log\log\log n)$. Comparing with Theorem 7.4, we see that the time bound is slightly better and that the structure of the algorithm is clearer for our direct construction.

8. Symmetric functions. A Boolean function is called symmetric if it depends only on the number of 1's in the input. In this section, we describe feasible algorithms with an almost optimal running time for computing symmetric functions. An obvious way to compute such a function is to count the 1's in the input and then to determine the function value depending on the count. Thus we start by studying the following task.

"*Sum of n bits*": For an input consisting of n bits $x_1, \ldots, x_n$ stored in n different memory cells, compute the binary representation of $\sum_{i=1}^{n} x_i$ and store it in $\|n\|$ memory cells (each bit in a separate cell).

We will describe a feasible algorithm for summing n bits and apply it to the task of evaluating symmetric functions in time $\varphi(n) + o(\log n)$ with n processors.

Remark 8.1. For the bit-summation problem, no circuits (with fan-in 2) of depth $\log n + o(\log n)$ are known. Thus the CREW PRAM algorithm described here is faster by a constant factor than what can be obtained by simulating the best known circuits for this problem. The same applies to the problem of computing arbitrary symmetric functions.

Let us first discuss two well-known methods, one for adding a sequence of bits and another for adding a sequence of binary numbers. Although these methods are not good enough for our purpose, we will need them as subroutines; moreover, our main algorithm is a generalization of the second method described. The first method that we will describe is based on a well-known very large-scale integration (VLSI) algorithm [21, §1.1.4]. It yields an algorithm for adding n bits that takes $O(\log n)$ steps on an EREW PRAM with n processors and n memory cells of wordsize 1.

LEMMA 8.2. *There is an n-processor EREW PRAM with n memory cells of wordsize* 1 *that computes the sum of* $n = 2^d$ *bits in* $4d$ *steps.*

Proof. We may assume that each READ phase consists of reading from two different cells. Such a machine, which we call a 2-READ PRAM, can be simulated by a usual EREW PRAM that makes twice as many steps. We prove the following by induction on k.

CLAIM. *There is a* 2*-READ PRAM* M_k *with* $2^k - 1$ *processors and memory cells* $C_0, \ldots, C_{2^k-1}$ *of wordsize* 1 *that for an input* $x_1, \ldots, x_{2^k}$ *writes the binary representation* $b_k b_{k-1} \cdots b_0$ *of* $\sum_{i=1}^{2^k} x_i$*, starting at step* k *with the least significant bit* b_0 *and writing* b_i *into cell* C_i *at step* $k+i$*. The cell* C_i *is not used by* M_k *after step* $k+i$*.* (*The total number of steps is* $2k$.)

M_1 reads the two input bits in step 1 and writes the two output bits in steps 1 and 2. Now we assume that M_k exists and use it to construct M_{k+1}. We split the 2^{k+1} input bits into two groups of 2^k bits each and use two copies M_k' and M_k'' of

M_k, one for each group. Let $C'_0, \ldots, C'_{2^k-1}$ and $C''_0, \ldots, C''_{2^k-1}$ be the cells used by M'_k and M''_k, respectively. By C_i we mean the cell C'_i if $i < 2^k$ or the cell C''_{i-2^k} if $i \geq 2^k$. In steps $t = k, k+1, \ldots, 2k$, the machines M'_k and M''_k produce the bits $b'_0, b'_1, \ldots, b'_k$ and $b''_0, b''_1, \ldots, b''_k$ of their (partial) sums. In addition, there is another processor P whose task it is to add up the two sums produced by M'_k and M''_k by using the standard "paper-and-pencil" method. Processor P keeps an internal variable c ("carry") initialized with 0; it is idle for the first k steps. In steps $t = k+1, \ldots, 2k+1$, processor P reads b'_i and b''_i for $i = t - (k+1)$, the values written by M'_k and M''_k in step $t-1$ into cells C'_i and C''_i, and adds b'_i, b''_i, and c, which results in a two-bit number $s_1 s_2$. Then P writes s_2 as the next output bit into cell C_i (the same cell as C'_i) and sets $c := s_1$ (s_1 is the next carry bit). In step $t = 2k+2$, processor P writes c. Obviously, P outputs the bits of $\sum_{i=1}^{2^k} x_i$ in the proper order and with the required timing. Altogether, $2(2^k - 1) + 1 = 2^{k+1} - 1$ processors are used. □

The second method that we describe is an adaptation of the "Wallace-tree" construction for a circuit of depth $O(\log u + \log k)$ for adding u k-bit numbers. This construction is based on the idea of "carry-save addition" [32, 33].

LEMMA 8.3.

(a) *If $b_i = b_{i,d-1} b_{i,d-2} \cdots b_{i,0}$ for $i = 1, 2, 3$ are three binary numbers with $b_1 + b_2 + b_3 < 2^d$, then there are two binary numbers $g_j = g_{j,d-1} g_{j,d-2} \cdots g_{j,0}$ for $j = 1, 2$ such that $b_1 + b_2 + b_3 = g_1 + g_2$ and such that there is an EREW PRAM of wordsize* 1 *with d processors that computes the bits $g_{j,e}$ from the bits $b_{i,f}$ in four steps.*

(b) *If u numbers b_i are given by their binary representations $b_{i,d-1} \cdots b_{i,0}$ for $0 \leq i < u$ and if $s = \sum_{i=0}^{u-1} b_i < 2^d$, then the binary representation $s_{d-1} \cdots s_0$ of s can be computed in $O(\log u + \log d)$ steps by an EREW PRAM with $d \cdot u$ processors and $d \cdot u$ cells of wordsize* 1.

Proof. (a) Let the sum $b_{1,2} + b_{2,l} + b_{3,l}$ for $0 \leq l < d$ have the binary representation $c_l s_l$. Define $g_{1,l} := s_l$ for $0 \leq l < d, g_{2,l} := c_{l-1}$ for $1 \leq l < d$, and $g_{2,0} := 0$. (Note that $c_{d-1} = 0$.) Obviously, $g_1 + g_2 = b_1 + b_2 + b_3$. The method for calculating the numbers $g_{j,l}$ on an EREW PRAM is obvious.

(b) The computation proceeds in stages. At the beginning of each stage, we have a collection of binary numbers with sum s. During a stage, we split these numbers into groups of three each, and then these three numbers are replaced by two new ones that have the same sum, as described in part (a). In that way, each stage reduces the number of the remaining numbers by a factor of $\frac{2}{3}$. Further, each stage needs at most $d \cdot u$ processors and $d \cdot u$ cells and takes four steps. These stages are performed until only two numbers are left. It is not hard to see that the number of stages does not exceed $1 + \log_{3/2} u$. Finally, to add up the two d-bit numbers that remain after the last stage, we use the algorithm of Corollary 7.4. The whole computation takes no more than $4 \cdot (1 + \log_{3/2} u) + O(\log d) = O(\log u + \log d)$ steps. □

Now we turn to the key result of this section.

THEOREM 8.4. *The sum of n bits can be computed by an n-processor CREW PRAM with n memory cells* (*of wordsize* 1) *in time $\varphi(n) + O(\log^{2/3} n \cdot \log\log n)$.*

Proof. Let $x_1, \ldots, x_n$ be the input bits, where, without loss of generality, $n = 2^d$. Our aim is to compute the binary representation of the sum $s = \sum_{i=1}^{n} x_i$ consisting of $\|n\|$ bits. The central idea of the algorithm is to generalize the addition method presented in Lemma 8.3 by using groups of more than three elements. Again, the algorithm proceeds in stages. The result of stage $i-1$ is a sequence $y_{i,1}, \ldots, y_{i,n_i}$ of binary numbers, each represented by $\|n\|$ bits, such that $\sum_{j=1}^{n_i} y_{i,j} = s$. The numbers $y_{i,j}$ are split into groups of a suitably chosen size a_i (as opposed to size three in

Lemma 8.3), and from the a_i numbers in each group a set of $\|a_i\|$ new numbers is computed that has the same sum. By collecting the new numbers from all groups, we obtain numbers $y_{i+1,1}, \dots, y_{i+1,n_{i+1}}$, where n_{i+1} is substantially smaller than n_i. We have to show how to perform one such stage efficiently, i.e., in almost optimal time, and how to choose the parameters a_i and n_i in such a way that the numbers n_i decrease so fast that not too many stages are necessary.

In addition, the algorithm has a preprocessing phase and a postprocessing phase. The preprocessing phase is necessary to get n_1 numbers $y_{1,1}, \dots, y_{1,n_1}$ with sum s, where n/n_1, the number of processors per cell, exceeds a minimal value necessary to start the main procedure. The postprocessing phase begins after stage m if n_m is sufficiently small. Then we add the remaining numbers using the procedure of Lemma 8.3(b). (We could also let the main procedure run until the end; however, this would unnecessarily complicate the analysis.) For the sake of simplicity, assume throughout the proof that n is a sufficiently large number. (For small n, the algorithm from Lemma 8.2 is used.) We start by describing an efficient method for transforming a numbers given in binary into $\|a\|$ new numbers that have the same sum.

LEMMA 8.5. *Let $a \geq 3$ and $n = 2^d \geq 1$ be integers. Then there is a CREW PRAM with $\|n\| \cdot a \cdot a^{\Phi(a)}$ processors and memory cells of wordsize* 1 *that, when presented with the binary representations of numbers $v_0, \dots, v_{a-1}$ with $\sum_{j=0}^{a-1} v_j \leq n$, computes in $\varphi(a) + 3$ steps the binary representations of numbers $z_0, \dots, z_{\|a\|-1}$ that satisfy*

$$\sum_{l=0}^{\|a\|-1} z_l = \sum_{j=0}^{a-1} v_j. \tag{8.1}$$

Proof. Let $v_{j,d} \cdots v_{j,0}$ be the binary representation of v_i for $j = 0, \dots, a-1$. For each bit position $c \in \{0, \dots, d\}$, consider the binary representation $b_{c,\|a\|-1} \cdots b_{c,0}$ of $b_c := \sum_{j=0}^{a-1} v_{j,c}$. We may rearrange these $\|a\| \cdot (d+1)$ bits so as to form $\|a\|$ numbers of length $\|n\| = d+1$ (see Figure 8.1 for an example). Namely, let z_l be the number with binary representation $b_{d-l,l} b_{d-l-1,l} \cdots b_{0,l} 0^l$. Then

$$\begin{aligned}\sum_{j=0}^{a-1} v_j &= \sum_{j=0}^{a-1} \sum_{c=0}^{d} v_{j,c} \cdot 2^c = \sum_{c=0}^{d} \left(\sum_{j=0}^{a-1} v_{j,c} \right) \cdot 2^c = \sum_{c=0}^{d} b_c \cdot 2^c \\ &= \sum_{c=0}^{d} \sum_{l=0}^{\|a\|-1} b_{c,l} \cdot 2^{l+c} = \sum_{l=0}^{\|a\|-1} \left(\sum_{c=0}^{d} b_{c,l} \cdot 2^{l+c} \right) = \sum_{l=0}^{\|a\|-1} z_l.\end{aligned}$$

(For the last equality, recall that $b_c \cdot 2^c \leq \sum_{j=0}^{a-1} v_j \leq n$, and hence $b_{c,l} = 0$ for $l+c > d$, i.e., for $c > d - l$.) Thus the numbers z_l satisfy equation (8.1).

By our definition, the binary representation of the numbers $z_0, \dots, z_{\|a\|-1}$ can be obtained by rearranging the bits of the binary representations of $b_0, \dots, b_d$. By Theorem 6.1 and Remark 6.2, each b_c can be computed in $\varphi(a) + 2$ steps by a CREW PRAM with $a \cdot a^{\Phi(a)}$ processors and cells of wordsize 1. Thus for $d+1 = \|n\|$ numbers b_c, we need $\|n\| \cdot a \cdot a^{\Phi(a)}$ processors (cells) altogether. In one additional step, the bits are rearranged so that we get binary representations of the numbers $z_0, \dots, z_{\|a\|-1}$. □

We now describe and analyze the algorithm for summing n bits. It consists of a preprocessing phase, a main procedure (in m stages), and a postprocessing phase.

$$
\begin{array}{cccccccccl}
 & & v_{05} & v_{04} & v_{03} & v_{02} & v_{01} & v_{00} & = & v_0 \\
 & & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & & \vdots \\
 & & v_{45} & v_{44} & v_{43} & v_{42} & v_{41} & v_{40} & = & v_4 \\
\hline
 & & & & & b_{02} & b_{01} & b_{00} & = & \text{Sum of last column} \\
 & & & & b_{12} & b_{11} & b_{10} & & = & \text{Sum of next to last column} \\
 & & & b_{22} & b_{21} & b_{20} & & & = & \cdots \\
 & & b_{32} & b_{31} & b_{30} & & & & & \cdots \\
 & b_{42} & b_{41} & b_{40} & & & & & & \cdots \\
\multicolumn{2}{c}{\underbrace{b_{52} \quad b_{51}}_{=0}} & b_{50} & & & & & & = & \text{Sum of first column} \\
 & & & & & \Downarrow & \text{Rearrange} & & & \\
 & & b_{32} & b_{22} & b_{12} & b_{02} & 0 & 0 & = & z_2 \\
 & & b_{41} & b_{31} & b_{21} & b_{11} & b_{01} & 0 & = & z_1 \\
 & & b_{50} & b_{40} & b_{30} & b_{20} & b_{10} & b_{00} & = & z_0
\end{array}
$$

FIG. 8.1. *Forming* $\|a\| = 3$ *numbers from* $a = 5$ *many by the procedure of Lemma* 8.5 *for* $d = 5$.

Preprocessing phase. The input bits are split into groups of 2^{w_1} elements with $w_1 := \lceil \log \|n\| \rceil + 4 \cdot \Phi(16)$. The sum of the bits in each group is computed using the algorithm of Lemma 8.2. As a result, we get $n_1 := 2^{d-w_1}$ numbers $y_{1,1}, \ldots, y_{1,n_1}$ of binary length $\|n\|$ (filling up with leading zeroes) such that $\sum_{i=1}^{n_1} y_{1,i} = s$. For the preprocessing phase, n processors, n cells, and $4w_1 = O(\log \log n)$ steps are sufficient.

Next, we describe stage i of the main procedure for $i \geq 1$. The number m of stages as well as the parameters $a_i = 2^{u_i}$ and $n_i = 2^{d-w_i}$ (where $u_i, w_i \in \mathbb{N}$) will be defined by recursion in the course of the description.

Stage i.

Input. The numbers $y_{i,1}, \ldots, y_{i,n_i}$ with $\sum_{j=1}^{n_i} y_{i,j} = s$. Each $y_{i,j}$ is given by a binary representation with $\|n\|$ bits.

Computation. The numbers $y_{i,j}$ are split into n_i/a_i groups of a_i elements each. Using the method of Lemma 8.5, each group is replaced by $\|a_i\|$ new numbers with the same sum.

Output. Let $n_{i+1} = 2^{\lceil \log \|a_i\| \rceil} \cdot n_i/a_i$. Output the numbers $y_{i+1,1}, \ldots, y_{i+1,n_{i+1}}$ obtained by collecting the $\|a_i\|$ results from each group padded with 0's to obtain n_{i+1} numbers.

LEMMA 8.6.

(a) $n_{i+1} < 2 \cdot \|a_i\| \cdot n_i/a_i$.

(b) *Stage i can be performed by $n_i \cdot \|n\| \cdot a_i^{\Phi(a_i)}$ processors and cells in $\varphi(a_i) + 3$ steps.*

Proof. (a) The proof of (a) is obvious. (b) Each of the n_i/a_i groups uses $\|n\| \cdot a_i \cdot a_i^{\Phi(a_i)}$ processors (cells). Hence $n_i \cdot \|n\| \cdot a_i^{\Phi(a_i)}$ processors (cells) are used altogether. □

We now define the numbers n_i and a_i for $i > 1$ (hence also w_i since $n_i = 2^{d-w_i}$). Assume that n_i has been determined already. By Lemma 8.6, $n_i \cdot \|n\| \cdot a_i^{\Phi(a_i)}$ processors (cells) are needed for performing stage i; thus we must have $n = 2^d \geq n_i \cdot \|n\| \cdot a_i^{\Phi(a_i)}$.

Define u_i as the largest number $u \in \mathbb{N}$ that satisfies $n \geq n_i \cdot \|n\| \cdot 2^{u \cdot \Phi(2^u)}$, and let $a_i = 2^{u_i}$. Note that $u_i \geq 4$ since $n/n_i = 2^{w_i} \geq 2^{w_1} \geq 2^{\log \|n\| + 4 \cdot \Phi(2^4)}$. The number m of stages of the main procedure is defined to be the minimal i such that $a_i \geq n_i$, i.e., $2^{u_i} \geq 2^{d-w_i}$, or $u_i + w_i \geq d$. In stage m, only one group of size n_m is formed; since $a_m \geq n_m$, the stage can be performed by the n processors. The result of the last stage is a set of $\|n_{m+1}\| \leq \|n\|$ numbers that have sum s.

Postprocessing phase. The algorithm of Lemma 8.3 is applied to the up to $\|n\|$ numbers of $\|n\|$ bits with sum s that result from the main procedure. This algorithm outputs the binary representation of s after $O(\log \|n\|) = O(\log \log n)$ steps using $\|n\| \cdot \|n\| = o(n)$ processors and cells.

It remains to analyze the main procedure. It is clear that n processors and memory cells are sufficient. The crucial step is to estimate m.

LEMMA 8.7. *The number m of stages of the main procedure is $O((\log n)^{2/3})$.*

Proof. From the construction, it is easily seen that the numbers u_i and w_i for $i \geq 1$ obey the following recursive definition:

$$\begin{aligned} w_1 &= \lceil \log \|n\| \rceil + 4 \cdot \Phi(16); \\ u_i &= \max\{u \mid \lceil \log \|n\| \rceil + u \cdot \Phi(2^u) \leq w_i\} \quad \text{for } i \geq 1; \\ w_{i+1} &= w_i + u_i - \lceil \log \|2^{u_i}\| \rceil \quad \text{for } i \geq 1. \end{aligned}$$

Since $w_i \geq w_1 = \lceil \log \|n\| \rceil + 4 \cdot \Phi(16)$, we have $u_i \geq 4$ for all i; hence $w_{i+1} \geq w_i + \frac{1}{4}u_i$ for all i since $\lceil \log \|2^u\| \rceil \leq \frac{3}{4}u$ for $u \geq 4$. It is not clear how to analyze the exact behavior of the sequence u_i, $i \geq 1$. Instead, we use a lower bound that is easier to handle. We recursively define a sequence v_i, $i \geq 1$, by

$$v_i = \max\left\{v \;\middle|\; 4v \cdot \Phi(2^{4v}) \leq 4 \cdot \Phi(16) + \sum_{j=1}^{i-1} v_j\right\} \quad \text{for } i \geq 1.$$

A straightforward induction on i shows that $v_i \leq u_i$ and $w_1 + v_1 + \cdots + v_{i-1} \leq w_i$ for all i. Since m is the minimal i such that $w_i + u_i \geq d$, it follows that

$$m \leq \min\{\, i \mid w_1 + v_1 + \cdots + v_i \geq d\}.$$

We apply Lemma 5.2 with $q = 1$, $A = 16$, $\eta = 0$, and $c = 4$ to the last expression, which yields the bound $m = O((d^2 \cdot 4)^{1/3}) = O((\log n)^{2/3})$. □

We may now finish the time analysis. For $i \leq m$, stage i of the main procedure takes $\varphi(a_i) + 3$ steps by Lemma 8.6(b). Altogether, then, stages 1 through m take $\sum_{i=1}^{m}(\varphi(a_i) + 3)$ steps. Both the preprocessing and postprocessing phases take

$O(\log\log n)$ steps. Thus the total computation time T can be estimated as follows using Lemmas 8.7 and 8.6(a):

$$\begin{aligned}
T &\leq \sum_{i=1}^{m}(\varphi(a_i)+3)+O(\log\log n)\\
&\leq \sum_{i=1}^{m}\log_b a_i + O(m) + O(\log\log n)\\
&\leq \sum_{i=1}^{m}(\log_b n_i - \log_b n_{i+1}) + \sum_{i=1}^{m}\log_b(2\|a_i\|) + O(\log^{2/3} n)\\
&\leq \log_b n_1 + m\cdot\log_b(2\|n\|) + O(\log^{2/3} n)\\
&\leq \varphi(n) + O(\log^{2/3} n\cdot\log\log n).
\end{aligned}$$

This completes the proof of Theorem 8.4. □

Theorem 8.4 is the key for computing symmetric functions with n processors fast. We need another auxiliary result.

LEMMA 8.8. *An arbitrary Boolean function F of k arguments can be computed by a CREW PRAM with 2^k processors and 2^k memory cells of wordsize* 1 *in time* $\varphi(k)+3$. *Moreover, after the READ phase of step* $\varphi(k)+3$, *there is a processor that knows the whole input string.*

Recall that a CREW PRAM with $k\cdot 2^{k-1}$ processors can compute F in time $\varphi(k)+1$. On the other hand, with only k processors for almost all Boolean functions, the computation takes $\Omega(k)$ steps provided that memory cells of wordsize 1 are used [4].

Proof. Let $k_1=\lfloor k/2\rfloor$ and $k_2=n-k_1$. We split the input string $x_1,\dots,x_k$ into substrings $v=x_1,\dots,x_{k_1}$ and $w=x_{k_1+1},\dots,x_k$ of length k_1 and k_2, respectively. Apply the algorithm of Fact 1.2 to v and w so that after the READ phase of step $\varphi(k_2)+1$, there is a processor P_v that knows v and a processor P_w that knows w. Note that $k_1\cdot 2^{k_1-1}+k_2\cdot 2^{k_2-1}\leq k\cdot 2^{k_2}<2^k$ processors (memory cells) have been used. We may assume that there are two sets $A_0,\dots,A_{2^{k_1}-1}$ and $B_0,\dots,B_{2^{k_2}-1}$ of memory cells, all initialized with 0. These cells may be used to code v and w: processor P_v writes a 1 into cell A_s, where s is the number with binary representation v, and processor P_w writes a 1 into cell B_r, where r is the number with binary representation w. (This is done during the WRITE phase of step $\varphi(k_2)+1$.) Since we have $2^k=2^{k_1}\cdot 2^{k_2}$ processors, we may assume that they are labeled $P_{i,j}$ for $0\leq i<2^{k_1}$, $0\leq j<2^{k_2}$. For each i and j, during the next two steps, processor $P_{i,j}$ reads cells A_i and B_j. Only processor $P_{s,r}$ reads a 1 both times. Then $P_{s,r}$ knows that the input string is $vw=x_1,\dots,x_k$ and may write the value $F(x_1,\dots,x_k)$ into the output cell during the WRITE phase of step $\varphi(k_2)+3\leq\varphi(k)+3$. □

THEOREM 8.9. *An arbitrary symmetric function of n arguments can be computed by an n-processor CREW PRAM with n memory cells* (*of wordsize* 1) *in time*

$$\varphi(n)+O(\log^{2/3}n\cdot\log\log n).$$

Proof. Let F be a symmetric function of n arguments. There is a function G of $k=\|n\|$ arguments such that if $y_1\cdots y_k$ is the binary representation of $\sum_{i=1}^{n}x_i$, then $F(x_1,\dots,x_n)=G(y_1,\dots,y_k)$. The machine that computes F first finds the sum of the input bits using the algorithm of Theorem 8.4. Thus it obtains k bits $y_1,\dots,y_k$ such that $F(x_1,\dots,x_n)=G(y_1,\dots,y_k)$. During the next two steps, all processors read the cells storing y_1 and y_2. Then the machine uses the algorithm of Lemma 8.8 for a function G_{y_1,y_2}, where $G_{i,j}(y_3, y_4, \dots, y_k) = G(i, j, y_3, y_4, \dots, y_k)$. The number of processors that we need for this phase is $2^{k-2} = 2^{\lceil \log(n+1) \rceil - 2} \leq 2^{\log(n+1)-1} = (n+1)/2$. Obviously, the total computation time is bounded by $\varphi(n) + O(\log^{2/3} n \cdot \log\log n) + O(\log\log n) = \varphi(n) + O(\log^{2/3} n \cdot \log\log n)$. □

9. Sorting algorithms.

In this section, we present algorithms for sorting n bits and sorting n binary numbers on small CREW PRAMs. In comparison to other sorting algorithms of logarithmic time complexity for networks and PRAMs [1, 8, 25], which sort arbitrary numbers but have a running time bound $C \cdot \log n$ with a constant C much larger than 1, the method below achieves optimal time up to a lower-order term for sorting bits. Finally, this method is extended to sorting arbitrary numbers given in binary representation.

THEOREM 9.1. *An n-processor CREW PRAM with n memory cells* (*of wordsize* 1) *can sort n bits in time* $\varphi(n) + O(\log^{2/3} n \cdot \log\log n)$.

Proof. The computation consists of three phases. During the first phase, the sum of the input bits is computed and written in binary. By Theorem 8.4, this can be done in $\varphi(n) + O(\log^{2/3} n \cdot \log\log n)$ steps. During the second phase, the algorithm of Lemma 8.8 is applied in order to get a processor that knows the sum of the input bits—say a number $s \leq n$. This phase takes $\varphi(\|n\|) + 3 = O(\log\log n)$ steps. During the third phase, in $O(\log\log n)$ steps, the number 1 is written into the cells $C_1, \dots, C_s$, and the number 0 is written into the cells $C_{s+1}, \dots, C_n$, thus getting the correct output. We describe the third phase in detail below. Given the time bounds claimed for the phases, it is clear that the whole computation takes at most $\varphi(n) + O(\log^{2/3} n \cdot \log\log n)$ steps.

The third phase consists of several stages. After each stage, we get the correct output values written into all cells except for a small block of decreasing size. If at the beginning of a stage, we start with an "unresolved" block B of size m, then during the stage the correct values are written into all cells of B except for some subblock of a size approximately equal to $\sqrt{m}$.

At the beginning of each stage, we have the following situation:

- There is a block of adjacent memory cells B and a number $z \in \mathbb{N}$ such that in order to get the correct output, the number 1 should be written into the first z cells of B and the number 0 into the rest of them.
- There are $|B|$ processors that know that B is an "unresolved" block, one processor associated with each cell of the block. There is one processor that knows the number z.
- All memory cells except for those in B already contain the correct output values.

We will describe only the first stage. The other stages are essentially the same, except for the last one, when the unresolved block has constant size and the output values are written sequentially by one processor. Without loss of generality, we may assume that $n = 2^d$. At the beginning of the first stage, the "unresolved block" consists of all cells. Let $D_1 = 2^{\lceil d/2 \rceil}$, $D_2 = 2^{\lfloor d/2 \rfloor}$, and $s = (r-1) \cdot D_1 + s_1$, where $0 < s_1 \leq D_1$ (hence also $r \leq D_2$ since $s \leq n = D_1 D_2$). We divide the memory cells $C_1, \dots, C_n$ into blocks of size D_1; namely, for $i \leq D_2$, block B_i consists of the cells $C_{(i-1)\cdot D_1+1}, C_{(i-1)\cdot D_1+2}, \dots, C_{i \cdot D_1}$.

Let P be the processor that knows s. In order to get D_2 processors that know r, the cells $C_1, \dots, C_{D_2}$ are cleared so that they contain 0's. Then processor P writes a 1 into cell C_r. Next, cells $C_1, \dots, C_{D_2}$ are read by the n processors, each cell by exactly D_1 of them. The processors that encounter a 1 know r. Then the blocks $B_1, \ldots, B_{r-1}$ are marked for filling with 1's and the blocks $B_{r+1}, \ldots, B_{D_2}$ for filling with 0's. Only block B_r might contain both 0's and 1's, so B_r cannot be filled at this stage of the computation. The marking of the blocks can be done in such a way that the symbol j is written into the first two cells of a block if this block should be filled with symbol j. The first two cells of block B_r receive 0 and 1 to indicate that this block should not be filled now. The marking can be done in two steps since we have $D_1 \geq D_2$ processors knowing r.

Then all n processors will be used again. For $i \leq n$, processor P_i reads the marked cells of the block containing cell C_i and, according to the situation, writes 0 or 1 into cell C_i or does not write if C_i lies in the block marked with 0 and 1.

This is the end of the first stage. Note that B_r is the "unresolved" block, that there are D_1 processors that know that B_r is the "unresolved" block (namely, the processors knowing r), and that processor P knows s and hence also s_1. To obtain the correct output, a 1 is to be written into the first s_1 cells of B_r and a 0 is to be written into the remaining cells of B_r. Hence we are in the correct situation for the beginning of the next stage.

Let $\gamma(m)$ be the time required by the procedure of the third phase for a block of m cells. By our construction, $\gamma(2^d) = \gamma(2^{\lceil d/2 \rceil}) + 6$. It follows that $\gamma(2^d) = O(\log d)$, i.e., $\gamma(n) = O(\log \log n)$, as required. □

Next, we show that binary numbers of arbitrary fixed length can be sorted in almost optimal time with small resources. As a preliminary step, we consider the problem of comparing two binary numbers.

FACT 9.2. *For each k, there is an EREW PRAM with k processors and $2k$ memory cells of wordsize 2 that compares two k-bit binary numbers in $\varphi(k)+2$ steps.*

Essentially, the algorithm for comparing two numbers is the same as for computing the logical OR. We briefly sketch the idea. Let $a = a_{k-1} \cdots a_0$ and $b = b_{k-1} \cdots b_0$ be two binary numbers. Define operators $\otimes$ and $\odot$ as follows. For $x, y \in \{0, 1\}$,

$$x \otimes y = \begin{cases} g & \text{if } x > y, \\ e & \text{if } x = y, \\ s & \text{if } x < y. \end{cases}$$

For $z, z' \in \{g, e, s\}$,

$$z \odot z' = \begin{cases} z' & \text{if } z = e, \\ z & \text{otherwise.} \end{cases}$$

Clearly, comparing a and b can be done by computing the product $z_{k-1} \odot z_{k-2} \odot \cdots \odot z_0$, where $z_i = a_i \otimes b_i$ for $i = 0, \ldots, k-1$.

After computing the symbols z_i in two parallel steps, the product $z_{k-1} \odot \cdots \odot z_0$ is computed using the same method as that for the logical OR [9]. This is possible since the only properties of the OR used are the associativity of the operator $\vee$ and that

$$x \vee y = \begin{cases} y & \text{if } x = 0, \\ x & \text{otherwise.} \end{cases}$$

(The difference is that the operator $\odot$ is defined over a domain of *three* elements.) We leave the details to the reader.

THEOREM 9.3. *There is a CREW PRAM with $m^2 \cdot k$ processors and $m \cdot (m+1) \cdot k$ memory cells of wordsize 1 that sorts m binary numbers of length k in time*

$$\varphi(m \cdot k) + O(\log^{2/3} m \cdot \log \log m).$$

Proof. Let k-bit numbers $a_1, \ldots, a_m$ be given. With each number a_j for $1 \leq j \leq m$, we associate a group G_j of $m \cdot k$ processors and $m \cdot k$ cells $C_{j,1}, \ldots, C_{j,m \cdot k}$. The processors of G_j determine the position that is to be taken by a_j in the sorted string and copy a_j to this place. This works as follows:

1. The processors of G_j copy the numbers $a_1, \ldots, a_m$ into the cells of G_j.

2. For each $i \leq m$, the number a_j is compared with a_i in $\varphi(k) + 2$ steps by k processors. Let

$$b_{j,i} = \begin{cases} 1 & \text{if } a_j < a_i \text{ or } (a_j = a_i \text{ and } j \geq i), \\ 0 & \text{otherwise.} \end{cases}$$

For each $i \leq m$, the number $b_{j,i}$ is written into cell $C_{j,i}$. (Note that the number of 1's in the string $b_{j,1}, \ldots, b_{j,m}$ determines the position of a_j in the sorted string.)

3. The bits $b_{j,1}, \ldots, b_{j,m}$ are sorted in $\varphi(m) + O(\log^{2/3} m \cdot \log\log m)$ steps; the resulting vector is written into cells $C_{j,1}, \ldots, C_{j,m}$.

4. For each $i < m$, there are k processors that read the cells $C_{j,i}$ and $C_{j,i+1}$. In that way, some k processors detect the last cell $C_{j,s}$ containing a 1. These processors copy the binary representation of a_j into cells $C_{s,1}, \ldots, C_{s,k}$.

Clearly, the cells $C_{1,1}, \ldots, C_{1,k}; C_{2,1}, \ldots, C_{2,k}; \ldots; C_{m,1}, \ldots, C_{m,k}$ contain the correct output after phase 4. Phases 1 and 4 require a constant number of steps, so the whole computation takes $\varphi(k) + \varphi(m) + O(\log^{2/3} m \cdot \log\log m)$ steps. However, $\varphi(k) + \varphi(m) = \log_b k + \log_b m + O(1) = \log_b(m \cdot k) + O(1) = \varphi(m \cdot k) + O(1)$. Hence the computation time is bounded as claimed. □

Acknowledgments. We thank Marcin Kik for many discussions and for pointing out that the algorithm of Theorem 6.1 can be used as a subroutine by the algorithm of Lemma 8.5. The constructive comments of two anonymous referees are gratefully acknowledged. The results on the simulation of circuits were found following a question of one of the referees.

REFERENCES

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

[2] P. Beame and J. Håstad, *Optimal bounds for decision problems on the CRCW PRAM*, J. Assoc. Comput. Mach., 36 (1989), pp. 643–670.

[3] P. Beame, M. Kik, and M. Kutyłowski, *Information broadcasting by exclusive-read PRAMs*, Parallel Process. Lett., 4 (1994), pp. 159–169.

[4] S. J. Bellantoni, *Parallel random access machines with bounded memory wordsize*, Inform. and Comput., 91 (1991), pp. 259–273.

[5] R. P. Brent, *The parallel evaluation of general arithmetic expressions*, J. Assoc. Comput. Mach., 21 (1974), pp. 201–208.

[6] J. Bruck, *Harmonic analysis of polynomial threshold functions*, SIAM J. Discrete Math., 3 (1990), pp. 168–177.

[7] S. Bublitz, U. Schürfeld, B. Voigt, and I. Wegener, *Properties of complexity measures for PRAMs and WRAMs*, Theoret. Comput. Sci., 48 (1986), pp. 53–73.

[8] R. Cole, *Parallel merge sort*, SIAM J. Comput., 17 (1988), pp. 770–785.

[9] S. Cook, C. Dwork, and R. Reischuk, *Upper and lower time bounds for parallel random access machines without simultaneous writes*, SIAM J. Comput., 15 (1986), pp. 87–97.

[10] A. Chandra, S. Fortune, and R. Lipton, *Unbounded fan-in circuits and associative functions*, J. Comput. System Sci., 30 (1985), pp. 222–234.

[11] M. Dietzfelbinger, M. Kutyłowski, and R. Reischuk, *Exact time bounds for computing Boolean functions on PRAMs without simultaneous writes*, in Proc. 2nd ACM Symposium on Parallel Algorithms and Architectures, Association for Computing Machinery, New York, 1990, pp. 125–135.

[12] ———, *Exact lower bounds for computing Boolean functions on CREW PRAMs*, J. Comput. System Sci., 48 (1994), pp. 231–254.

[13] F. Fich, *New bounds for parallel prefix circuits*, in Proc. 15th Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1983, pp. 100–109.

[14] F. Fich and A. Wigderson, *Towards understanding exclusive write*, SIAM J. Comput., 19 (1990), pp. 718–727.

[15] R. M. Karp and V. Ramachandran, *Parallel algorithms for shared-memory machines*, in Handbook of Theoretical Computer Science, Vol. A, Algorithms and Complexity, J. van Leeuwen, ed., Elsevier, Amsterdam, 1990, pp. 869–941.

[16] M. Kutyłowski, *Fast algorithms for threshold functions on CREW PRAMs*, Technical Report, University of Wrocław, Wrocław, Poland, 1991.

[17] ———, *Time complexity of Boolean functions on CREW PRAMs*, SIAM J. Comput., 20 (1991), pp. 824–833.

[18] M. Kutyłowski and R. Reischuk, *Evaluating formulas on parallel machines without simultaneous writes*, Technical Report, Institut für Theoretische Informatik, Technische Hochschule Darmstadt, Darmstadt, Germany, 1990.

[19] R. E. Ladner and M. J. Fischer, *Parallel prefix computation*, J. Assoc. Comput. Mach., 27 (1980), pp. 831–838.

[20] K. Lange, *Unambiguity of circuits*, Theoret. Comput. Sci., 107 (1993), pp. 77–94.

[21] F. T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*, Morgan Kaufmann Publishers, San Mateo, CA, 1991.

[22] N. Nisan, *CREW PRAMs and decision trees*, SIAM J. Comput., 20 (1991), pp. 999–1007.

[23] N. Nisan and M. Szegedy, *On the degree of Boolean functions as real polynomials*, Comput. Complexity, 4 (1994), pp. 301–313.

[24] I. Parberry and P. Y. Yan, *Improved upper and lower time bounds for parallel random access machines without simultaneous writes*, SIAM J. Comput., 20 (1991), pp. 88–99.

[25] M. Paterson, *Improved sorting networks with $O(\log n)$ depth*, Algorithmica, 5 (1990), pp. 75–92.

[26] J. H. Reif, ed., *Synthesis of Parallel Algorithms*, Morgan Kaufmann Publishers, San Mateo, CA, 1993.

[27] R. Smolensky, *Algebraic methods in the theory of lower bounds for Boolean circuit complexity*, in Proc. 19th Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1987, pp. 77–82.

[28] M. Snir, *On parallel searching*, SIAM J. Comput., 14 (1985), pp. 688–708.

[29] L. Stockmeyer and U. Vishkin, *Simulation of parallel random access machines by circuits*, SIAM J. Comput., 13 (1984), pp. 402–422.

[30] M. Szegedy, *Algebraic Methods in Lower Bounds for Computational Models with Limited Communication*, Ph.D. thesis, Department of Computer Science, University of Chicago, Chicago, 1989.

[31] U. Vishkin and A. Wigderson, *Trade-offs between depth and width in parallel computation*, SIAM J. Comput., 14 (1985), pp. 303–314.

[32] C. S. Wallace, *A suggestion for a fast multiplier*, IEEE Trans. Comput., 13 (1964), pp. 14–17.

[33] I. Wegener, *The Complexity of Boolean Functions*, Wiley–Teubner, Stuttgart, 1987.

SIAM J. COMPUT.
Vol. 25, No. 6, pp. 1231–1253, December 1996

LOWER BOUNDS FOR GEOMETRICAL AND PHYSICAL PROBLEMS*

JÜRGEN SELLEN[†]

Abstract. Motion planning involving arbitrarily many degrees of freedom is known to be PSPACE-hard. In this paper, we examine the complexity of generalized motion-planning problems for planar mechanisms consisting of independently movable objects.

Our constructions constitute a general framework for reducing problems in information processing to motion planning, leading to easy proofs of known PSPACE-hardness results and to exponential lower bounds for geometrical problems related to motion planning. Particulalrly, we show that the problem of deciding whether a given mechanism A can always avoid a collision with another mechanism B is EXPSPACE-hard.

New lower bounds are also obtained for the problem of planning under given physical side conditions. We consider the case that certain motions require forces, e.g., to subdue friction, and ask for motions that stay under a given energy limit. Within our framework, we show that such shortest-path problems are EXPTIME-hard if we use number representations by mantissa and exponent, and even undecidable if we allow that some motions require no force or an infinite amount. The proof consists of a simulation of Turing machines with infinite tape and shows that the notion of Turing computability can be interpreted in purely geometrical terms. The geometrical model obtained is capable of expressing a variety of physical-planning problems.

Key words. motion planning, lower bounds, kinematic problems, dynamic problems, geometrical problems, collision-avoidability problem

AMS subject classifications. 68Q05, 03D10, 68Q25, 03D15, 68U05, 65Y25

1. Introduction.

This paper is concerned with motion-planning and related problems with and without physical (resp., dynamic) constraints. In our terminology, we use the notion *kinematic scene* to describe the physical world which constitutes the input of a kinematic problem. A kinematic scene is a collection of rigid objects, each object being a semialgebraic, connected subset of the Euclidean space $\mathbf{R}^d$ $(d = 2, 3)$, described by a Boolean expression of polynomial inequalities of the form $p(x) > 0$ or $p(x) \geq 0$ with integer coefficients. Each object is specified as either an obstacle, which has no freedom of motion, or a movable object, the possible motions of which correspond to the Euclidean group of rigid transformations. The complexity of a kinematic scene is the length of this description with the integers coded in binary if not further specified.

With this definition, the classical motion-planning problem can be stated as follows: given a kinematic scene and the initial and final placements of the objects, find a motion connecting these placements or decide that no such motion exists.

A standard formalism for specifying positions and motions is based on the notions *configuration space* and *free space.* By configuration space CS, we denote the topological space which is spanned by all of the parameters that uniquely determine the positions of the movable objects. The subspace consisting of the physically legal configurations is called the free space FS of the kinematic scene. We define the *legal configurations* to be those in which the transformed objects, considered as point sets in $\mathbf{R}^d$, have a pairwise-empty intersection. An allowed motion in a scene corresponds to a path in its free space, i.e., a continuous function $w : [0, 1] \rightarrow \text{FS}$. For motion

* Received by the editors January 26, 1993; accepted for publication (in revised form) February 14, 1995. This research was supported by a Graduiertenkolleg-fellowship of the Deutsche Forschungsgemeinschaft.

[†] Fachbereich 14 Informatik, Universität des Saarlandes, Postfach 151150, 66041 Saarbrücken, Germany (sellen@cs.uni-sb.de).

planning, we are only interested in the connected (short for "path-connected") component of the free space of a given scene S that contains an initial configuration C_0, and we shall further denote this subspace by $\mathrm{FS}_{C_0}(S)$.

General solutions to the motion-planning problem are based on the fact that the free space can be described as a semialgebraic subset of a k-dimensional Euclidean space, where k is proportional to the total number of the degrees of freedom (i.e., summarized over all independently movable objects). By decomposing the free space into connected cells and interpreting these cells as nodes of an adjacency graph, motion planning can be reduced to a simple path search in a graph. Using Collins's algorithm of cylindrical algebraic cell decomposition, Schwartz and Sharir developed an algorithm whose running time is doubly exponential in dimension k [1, 2]. This was improved by Canny, who proposed a single exponential algorithm based on a retraction technique [3]. In [4], Canny showed that the motion-planning problem can be solved in polynomial space.

These algorithms seem to be worst-case optimal since there are kinematic scenes of complexity n with k movable objects whose free space consists of n^k connected components. It should be noted that n^k also forms an upper bound for the complexity of the free space: combinatorial arguments show that the number of all faces comprising the boundary of the free space is at most n^k high [5].

In fact, the decision problem for motion planning involving arbitrarily many degrees of freedom can be shown to be PSPACE-hard for even very restricted subclasses of the problem.

Reif showed that motion planning is PSPACE-hard if we restrict our considerations to the motion of one object consisting of many linked parts in a three-dimensional tunnel system [6]. Hopcroft, Joseph, and Whitesides investigated planar linkages as a simple model for robots, i.e., systems of rods whose endpoints are connected by joints or attached to the plane. Using the classical result that any multivariate polynomial $p(x_1, \ldots, x_n)$ can be "represented" by a planar linkage, they showed that it is possible to construct for any space-bounded Turing machine T a planar linkage capable of simulating T [7]. Searching for a moving system whose geometry is as simple as possible but for which motion planning is still intractable, Hopcroft, Schwartz, and Sharir proved that motion planning for multiple independent rectangular boxes sliding inside a rectangular box is PSPACE-hard [8]. However, the slightly easier problem of moving many discs inside a planar polygonal world could only be proven to be strongly NP-hard [9].

In this paper, we present a general framework for performing sound and transparent translations of known problems in information processing to motion planning, which will provide us with exponential lower bounds for some geometrical problems related to motion planning as well as for planning problems in extended models involving physical side conditions.

This framework is based on the construction of mechanisms that are able to process information in the context of motion planning. Planar kinematic scenes that consist of polygonal objects with purely translational freedom of motion turn out to be powerful enough to provide us with most of the necessary tools. In §2, we construct scenes for elementary tasks such as storage of information, Boolean operations, or arithmetic on real numbers and show how these "basic mechanisms" can be combined to obtain "devices" for more complicated tasks, e.g., binary counters.

The generality of this framework and the clarity of reductions follows from the close relationship to circuit design. Nevertheless, we do not build "mechanical computers" as Zuse did in the first half of this century. To express information processing as motion planning, the existence of some motion has to correspond to some condition, but we do not consider static states or acting forces to generate these motions deterministically.

In §3, we prove exponential lower bounds for some kinematic problems in the field of motion planning. These problems also possess interesting geometrical interpretations. We show that the problem of deciding whether a quantified query for allowed configurations has a positive or negative answer is EXPTIME-hard and that the problem of deciding if a given mechanism A can always avoid a collision with another given mechanism B is EXPSPACE-hard. The proof of the first result is based on the fact that a restricted arithmetic on real numbers can be simulated by motion planning.

In practical applications, motions usually have to fulfill criteria other than simply being free of collisions. Since such restrictions seem to add a new quality to the problem, the robotics community pays increasing attention to problems involving physical side conditions. These problems belong to the general framework of nonholonomic motion planning. In [10], Fortune and Wilfong describe an algorithm for planning motions such that the corresponding path in free space is continuously differentiable and of bounded curvature (see also Jacobs and Canny [11]). In [12], motions are required to be time optimal and to have bounded velocity and acceleration ("kinodynamic motion planning"). For both cases, only exponential, exact solutions are known, though the hardness is still open. However, the problem of finding a shortest path between two points in a three-dimensional polygonal environment is already NP-hard [13]. A survey of nonholonomic motion planning can be found in [14].

In §4, we examine the fact that motions consume energy and that some motions may be more expensive in this sense than others (e.g., due to friction). We use the notion *dynamic scene* for a kinematic scene in which a cost function is associated with the possible motions of each object depending on its contact situation. To obtain a sound definition of the input of our problems, we shall further assume that the cost functions are given implicitly by assigning integer friction coefficients to the sides of the objects. The interesting problem here is to find motions with minimal cost or to decide if there exists a motion that stays under a given cost limit. For these problems, we prove exponential lower bounds. If we allow that some motions are forbidden or cost nothing, we even get undecidability. (This was first discovered by Reif; see [6].) Undecidability is also obtained for the similar problem of deciding if there exists a specific motion in a kinematic scene during which not more than a given constant number of objects is moved simultaneously.

These results follow from a simulation of Turing machines with unbounded or exponentially bounded tape by motion planning in dynamic scenes. The inferences of this simulation technique are of interest from both a complexity-theoretic and a practical point of view. A translation of the dynamic scenes underlying the simulation into the formalism of configuration space provides us with a geometrical interpretation of Turing computability. At the same time, we obtain a geometrical model in which a variety of physical problems can be expressed.

Although the lower bounds that we obtain are valid only for problem instances that involve arbitrarily many degrees of freedom, our work should cast some light on the common structure and complexity of physical and geometrical problems and should provide a framework for further research.

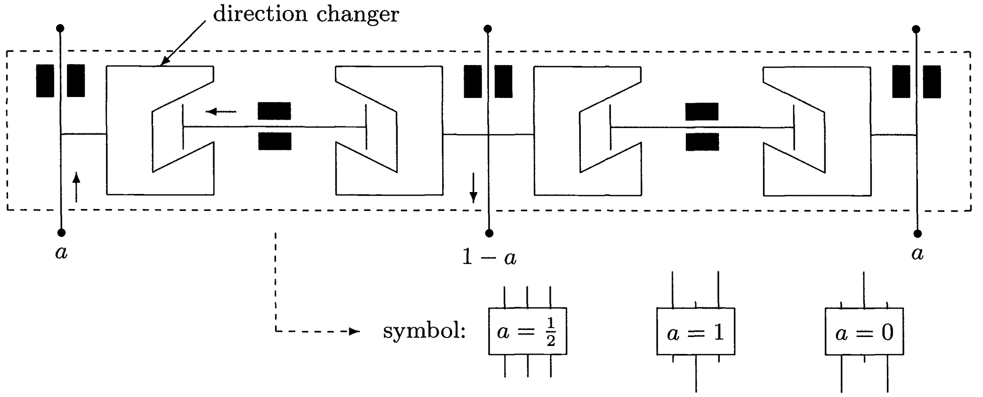


FIG. 1. *A variable cell. The position of the vertical rod at the left corresponds to the value of the variable a. By moving this rod up, the horizontal rod is pulled to the left, and the vertical rod in the middle, which represents* $1 - a$*, is pushed downward.*

2. Information processing by planar mechanisms.

In this section, we construct tools for processing information in planar kinematic scenes. We show how information can be coded, stored, and transported and explore which operations can be realized.

Our constructions are illustrated by Figures 1–9. The filled boxes correspond to obstacles. All other objects are movable. For clarity, rigid connections between rods are sometimes marked by fat points. We recall that the sides of our objects may be "closed" or "open" depending on whether they are described by an inequality $p(x) > 0$ or $p(x) \geq 0$. Although this is not marked in the figures, we make use of both possibilities and leave it to the reader to complete the drafts.

2.1. Coding of information.

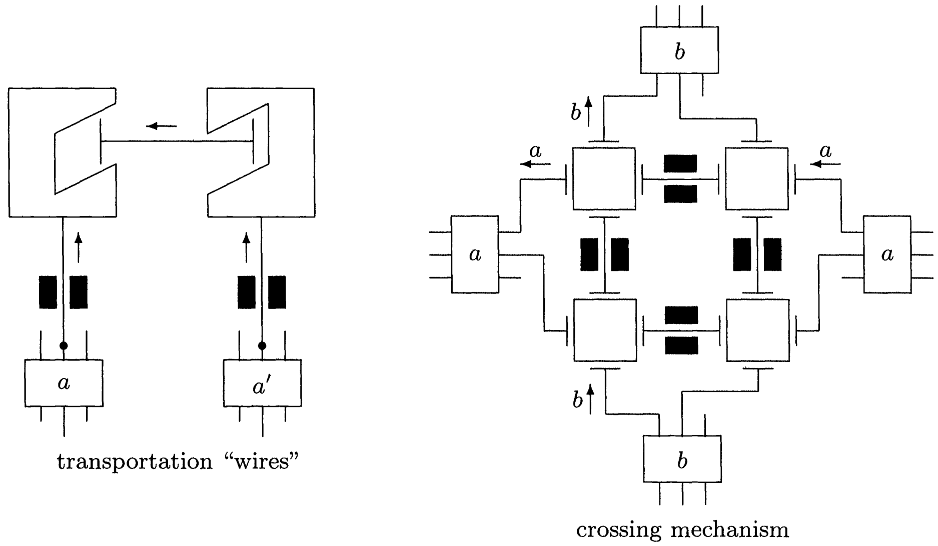
To represent logical and numerical information, we code a number $a \in [0, 1]$ as the vertical position of a rod whose freedom of motion is restricted by obstacles to an interval of length 1 in the vertical direction. The logical values *true* and *false* correspond to the values 1 and 0.

For processing purposes, it is useful to have a second rod attached to the first representing the inverse value $1 - a$ and, to obtain symmetry, a third rod again representing a. Thus we get a small mechanism whose configuration corresponds to the value of a variable a and which we shall denote as *variable cell*, symbolized by a box with label a (see Figure 1). The coupling of the three rods is realized by subassemblies transforming vertical movements to horizontal movements and vice versa. These subassemblies constitute a basic element of our constructions, and we shall denote them as *direction changers*.

2.2. Transportation of information.

In order to perform operations on variables, we need the possibility of transporting the actual value of a variable from any place in the plane to any other place. Figure 2 shows how this can be achieved by a mechanism that forces equality between the values of two arbitrarily positioned variable cells a and a': the cells are connected by a sequence of rods and direction changers. Analogous to circuit design, we denote such sequences as *wires*, but we remark that they consist of movable rods carrying information by their position.

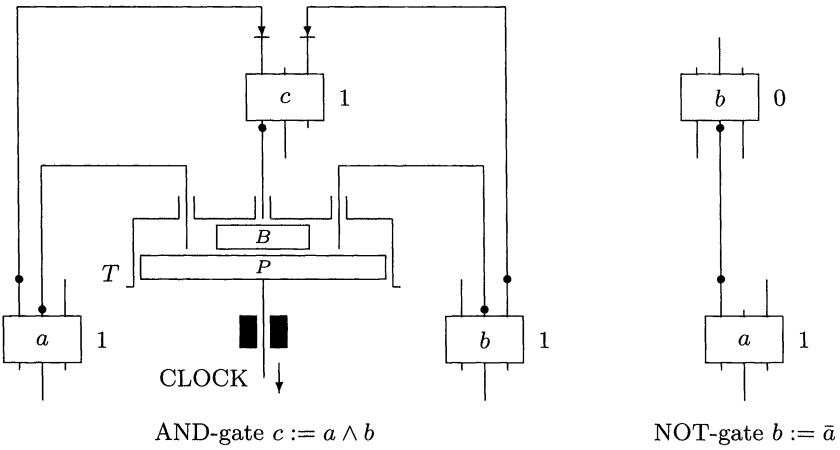
In order to obtain planar constructions, we need the possibility of crossing wires. This can be achieved by exploiting the fact that in planar scenes, objects have two



transportation "wires"

crossing mechanism

FIG. 2. *Wires and crossings. If, in the crossing mechanism, b is increased, then the two left boxes are pulled upward, and simultaneously the two right boxes are pushed downward (and analogously for variable a, whose cells are rotated by* $90°$*).*



AND-gate $c := a \wedge b$

NOT-gate $b := \bar{a}$

FIG. 3. *Mechanisms that realize the Boolean operations NOT and AND. In the AND-gate, the position of P shown corresponds to $CLOCK = 1$.*

degrees of translational freedom and thus can carry two independent information values at the same time. Figure 2 shows how two horizontal wires that represent a and $1 - a$ can be crossed with two vertical wires that represent b and $1 - b$. This is done by four boxes that carry both informations: the horizontal positions of the boxes are uniquely determined by the value of a and the vertical positions by the value of b. Changes in the value of variable a (resp., b) affect only the horizontal (resp., vertical) positions of the boxes and thus leave the value of variable b (resp., a) unchanged.

2.3. Logical operations. Planar kinematic scenes are capable of evaluating arbitrary Boolean expressions in the sense that, given an expression $b(x_1, \ldots, x_n)$, there

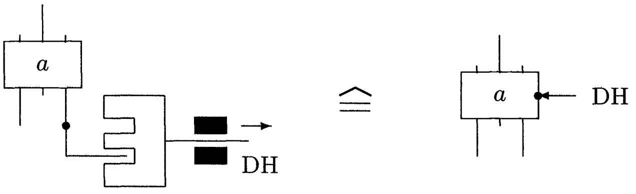


FIG. 4. *A binary memory cell. If the comb is in its leftmost position, corresponding to DH* = 1, *the value of a is fixed at either* 0 *or* 1.

is a scene with variable cells for $x_1, \ldots, x_n$ and x in which some specified motion forces the value of x to be $b(x_1, \ldots, x_n)$ if $x_1, \ldots, x_n$ carry fixed logical values. We obtain this by constructing *mechanical gates*, shown in Figure 3, for the Boolean operations NOT and AND.

The negation $a := \bar{b}$ can be achieved by connecting a rod coding a with a rod coding $1 - b$. To realize $c := a \wedge b$, we need a clock signal, i.e., a motion of an additional object that forces c to be $a \wedge b$. The position of this object, a vertically movable "plunger" P, is further denoted as CLOCK.

To check the correctness of the construction in Figure 3, let us assume that the plunger is in its upmost position, corresponding to CLOCK = 1. The arrows over the rods that code c are connected to the rods that code a and b and may press the rods that code c downward. If $a = 1$ and $b = 1$, then these arrows impose no restriction on the value of c, but the small box B must be in a middle position, forcing c to be 1. If $a = 0$ or $b = 0$, then the box B can be on the left or right side of the "tub" T and hence does not affect c, but at least one of the arrows forces c to be 0.

We need the CLOCK since the values of a and b can not be changed while the plunger stays in its upmost position. Only if the plunger is in its downmost position, corresponding to CLOCK = 0, can the variables a and b be "adjusted" arbitrarily.

2.4. Storage and copying of information. The constructions of the previous subsections show that combinational logic circuits can be simulated by motion-planning problems in kinematic scenes. However, to simulate sequential circuits, we still have to introduce the possibility of storing information.

Figure 4 shows how a memory cell that is able to store the logical values 0 and 1 can be realized. We again use a clock signal, denoted as DATA HOLD (DH), to force the cell to keep its information. The value of DH corresponds to the position of a horizontally movable "comb" that can hold a rod of a variable cell in a position that codes 0 or 1.

It is possible to construct cells of polynomial complexity that are able to store a single exponential number of different values. Nevertheless, the upper bound on the combinatorial complexity of free space shows that storing a doubly exponential amount of information cannot be achieved in the model of kinematic scenes.

We are now able to store information. To process information as in sequential circuits, we still need to realize loops or feedbacks. We need the possibility of copying information, i.e., forcing equality between the values of two variable cells according to some clock signal.

A possible realization of such a copy operation is shown in Figure 5. By moving two rods between the rods that code a / $1-a$ and b / $1-b$, the values of variables a and b are coupled. The two rods are connected in the horizontal direction such that their

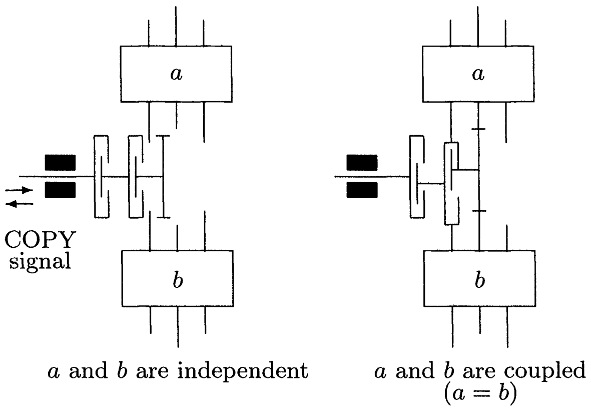


FIG. 5. *A copy mechanism. The COPY signal determines the horizontal position of three parts which can be vertically shifted independently of each other.*

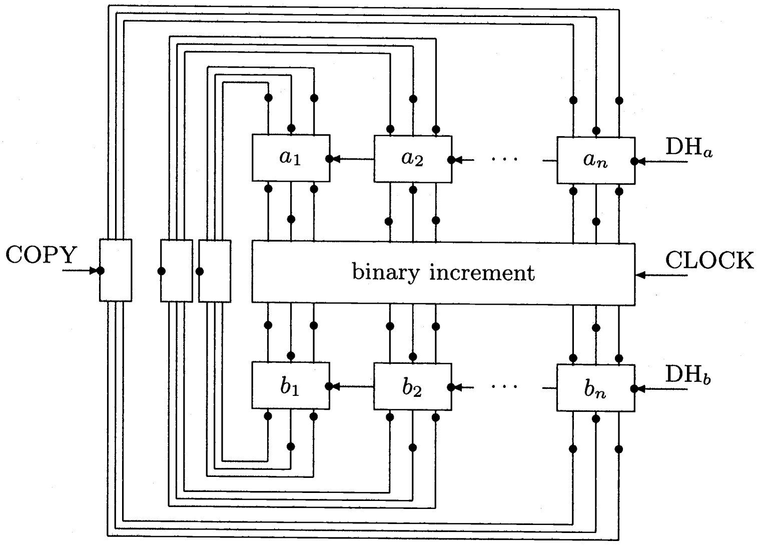
relative position is constant but such that they are free to slide independently from each other in the vertical direction. Their absolute horizontal position is determined by the value of the clock, which we shall further denote as the COPY signal.

2.5. Sequential circuits and control sequences. The constructions of the previous subsections provide us with most of the tools that are necessary to simulate sequential circuits. The only problem that remains to be solved is the generation of adequate sequences of clock signals.

We demonstrate the construction of a scene simulating a sequential circuit with the example of a binary counter. The mechanism is illustrated in Figure 6 and consists of (i) two chains of binary memory cells for storing the numbers a and b according to the DH signals DH_a and DH_b, (ii) a logic circuit that forces b to be $a + 1$ if the CLOCK signal is set to 1, and (iii) a chain of copy mechanisms that force a and b to be equal if the COPY signal is set to 1. Counting can then be forced by cyclically repeating the listed signal sequence.

Figure 7 shows how a cyclic repetition of a signal sequence can be generated. A box can be moved cyclically inside a "ring" that consists of eight rectangular "chambers." Each chamber can be individually forced to be empty ("closed") by pressing a plunger inside. To achieve the situation where only two chambers are open at the same time, there are mechanisms connecting the plungers which, if one chamber is open, force all other chambers except the two neighboring ones to be closed. Each chamber corresponds to a row in the signal table: the plunger for opening and closing chamber i is connected with the signal wires of the signals that are registered in row i as 1 such that these signals are set to 1 by opening the chamber. By moving circularly inside the ring, the box thus generates a cyclical repetition of the signal sequence. (Since two chambers may be open at the same time, the "active" signals of the two corresponding rows may be forced to be 1 simultaneously, but this does not affect the simulation process for the chosen signal table.)

We conclude that planar kinematic scenes are capable of simulating arbitrary sequential circuits in the context of motion planning. Particularly, we can simulate PSPACE-bounded Turing machines by realizing the tape and state as chains of binary memory cells. Since the construction of the adequate kinematic scene can obviously be performed in polynomial time, we obtain a new proof of the PSPACE-hardness of



	DH$_a$	CLOCK	DH$_b$	COPY
1	1	0	0	0
2	1	1	0	0
3	1	1	1	0
4	0	0	1	0
5	0	0	1	1
6	1	0	1	1
7	1	0	0	0
8	1	0	0	0

FIG. 6. *A binary counter. Counting is forced by cyclically repeating the given signal sequence.*

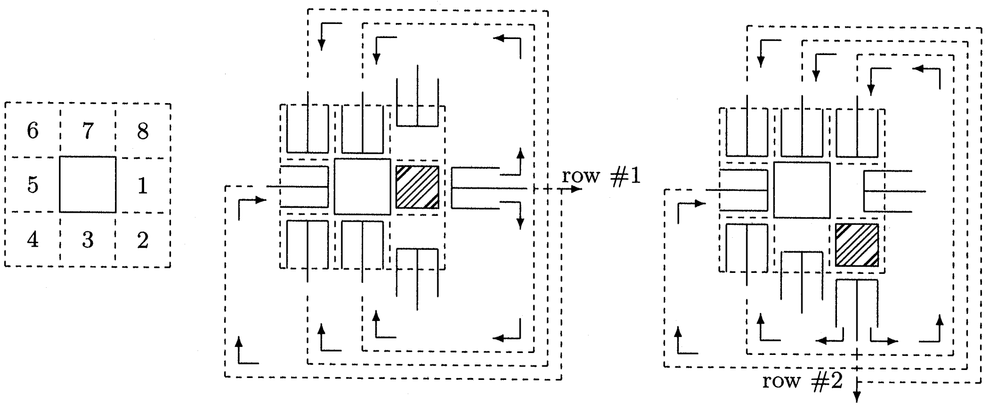
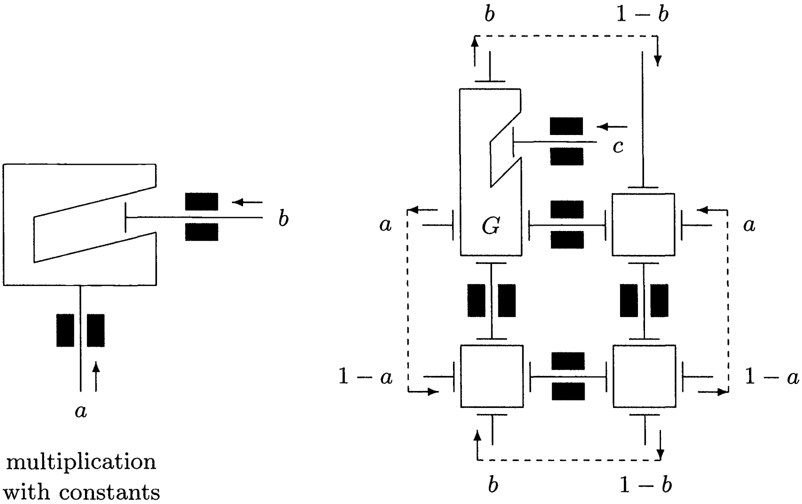


FIG. 7. *A mechanism for producing a cyclical repetition of a given signal sequence. The dashed lines represent the mechanisms connected to the plungers that open or close the chambers.* (*Only the mechanisms connected to plungers* 1 *and* 2 *are indicated.*)



multiplication with constants $b := c \cdot a$

addition $c := a + b$

FIG. 8. *Mechanisms that realize the arithmetic operations multiplication with constants and addition.*

the classical motion-planning problem.

The reader should note that it is only possible to simulate symmetric Turing machines. Since motions can always be reversed (in our construction, the "direction" of the cyclical signal sequence can always change), the operations of the simulated machines can also be carried out in reverse order. This is captured by the notion of symmetric Turing machines and does not affect our complexity results [15].

2.6. Arithmetic on real numbers. Although we are able to code real numbers, we introduced only mechanisms for dealing with logical values. In this subsection, we construct kinematic scenes that are able to perform arithmetic operations, providing us with the possibility of describing problems in the theory of real numbers with motion-planning problems.

In Figure 8, mechanisms that realize the operations of multiplication with constants ($b := c \cdot a$) and addition ($c := a + b$) are shown. Multiplication with constants can be achieved by using specially designed direction changers: the slope in the notch of the changer corresponds to the constant c. The mechanism for the addition of the two numbers a and b is based on our mechanism for realizing crossings. We again cross the wires that carry the values of a / $1 - a$ and b / $1 - b$ by four boxes. The box whose horizontal position corresponds to a and whose vertical position corresponds to b (i.e., box G) contains a notch in which a rod that codes the result c grips. This rod is pulled horizontally to the left both if the wire for a is moved to the left (i.e., a is increased) and if the wire for b is moved upward (i.e., b is increased). The position of this rod thus corresponds to the value of the addition result. Obviously, subtraction can also be realized by this mechanism if we render the notch of G.

The free spaces of kinematic scenes that consist of polygonal objects with only translational freedom of motion are multidimensional polyhedra. Thus it is not possible to realize "nonlinear" operations like multiplication of two real numbers by such scenes. However, we remark without proof that multiplication can be realized by planar linkages and thus by scenes in three-dimensional space that consist of objects with rotational degrees of freedom [7, 16].

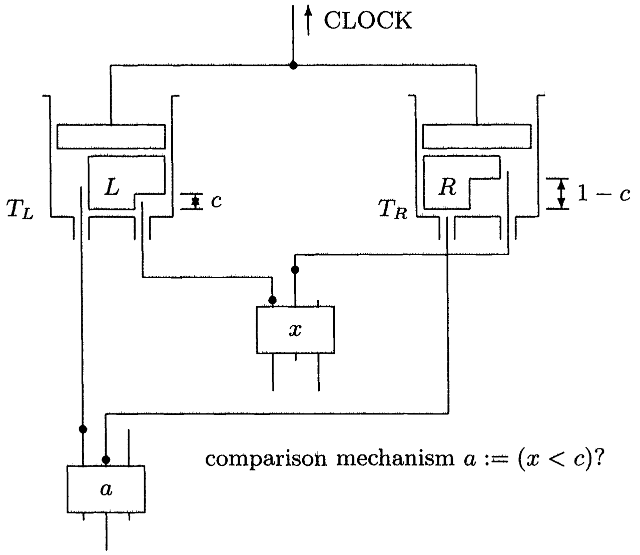


FIG. 9. *A mechanism that realizes the comparison* $a := (x < c)?$.

To combine the capabilities for dealing with logical and real values, e.g., for evaluating Boolean expressions of polynomial inequalities, we must be able to perform comparisons with costants. Figure 9 shows a clocked mechanism that realizes the comparison $a := (x < c)?$. If cell x carries a fixed real value from the domain $[0, 1]$, then the value of cell a is forced to be 1 if $x < c$ and 0 if $x \geq c$ by moving the object that corresponds to the CLOCK signal into the position for CLOCK $= 1$. The adjustment of a is forced by boxes L and R, whose shape is determined by the constant c. If CLOCK $= 1$ and $x \geq c$, then box L must be in the left area of tub T_L, which forces a to be 0. If $x < c$, then $1 - x > 1 - c$ and box R must be in the left area of tub T_R, which forces a to be 1. The CLOCK is again needed to have the possibility of readjusting x.

3. Exponential lower bounds for kinematic problems. In this section, we present two decision problems within the model of kinematic scenes for which we can prove exponential lower bounds using the constructions of the previous section.

The first problem concerns the avoidability of collisions between two given mechanisms A and B. For any possible motion of B (the "attacker") decide if there exists a simultaneous motion of A (the "defender") such that both mechanisms do not collide. A practical application is to decide whether the motions of two given autonomous robot systems can always be coordinated such that one system can perform all its tasks in the presence of the other. We formulate the problem more geometrically as a *path-lifting problem* in the configuration space of a kinematic scene.

PROBLEM 1 (collision-avoidability problem (CAP)). *Given a planar kinematic scene S that consists of two disjoint subscenes S_A and S_B (i.e., the object set of S is the disjoint union of the object sets of S_A and S_B) and an initial configuration $C_0 \in \mathrm{FS}(S)$, decide if, for any path w in the free space $\mathrm{FS}(S_B)$ that starts at $C_0|_{S_B}$, there exists a path w' in $\mathrm{FS}(S)$ that starts at C_0 whose projection to $\mathrm{FS}(S_B)$ is equal to w.*

We shall prove that this problem cannot be solved in polynomial time or space

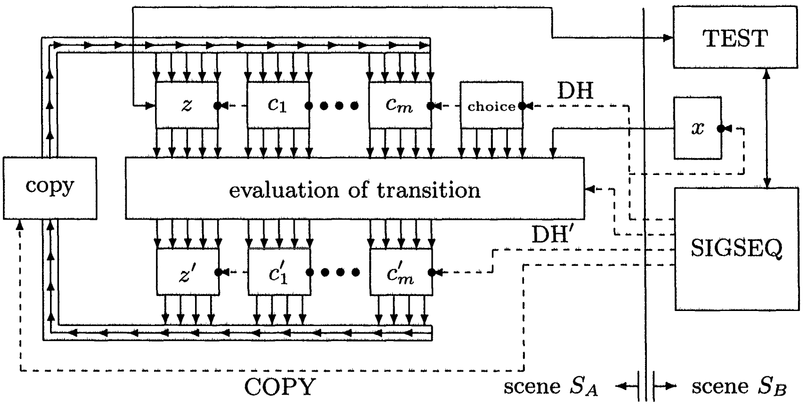


FIG. 10. *Assembly of a scene for the CAP. Control signals are indicated by dashed lines.*

by reducing to it the following problem L_{rex}, which is known to be EXPSPACE-hard [17]:

Given a regular expression r with exponentiation (exponents are coded in binary), decide if the generated language $L(r)$ is equal to X^*, where X denotes the alphabet. (An example is $L((\text{a}|\text{b}^3)^2) = \{\text{aa,abbb,bbba,bbbbbb}\}$.)

THEOREM 3.1. *The CAP is EXPSPACE-hard.*

Proof ($L_{\text{rex}} \leq$ CAP). For any given regular expression r with exponentiation, we construct in polynomial time a kinematic scene S consisting of two subscenes S_A and S_B such that S_A simulates an automaton that accepts $L(r)$ and S_B produces arbitrary input words for S_A. The structure of S is drafted in Figure 10.

Step 1: *Construction of an automaton $\mathcal{A}$ that accepts $L(r)$.* The number of states of a finite automaton accepting $L(r)$ may grow exponentially in the length of r and even double exponentially if we require the automaton to be deterministic. However, we can construct a nondeterministic automaton of polynomial size that accepts $L(r)$ by providing the state set with a finite memory that consists of counters, one for each exponent. We obtain an automaton with state set $Q = Z \times C_1 \times \cdots \times C_m$, where Z denotes a usual finite state set and $C_i = \{0, 1, \ldots, n_i\}$ the domain of the ith counter. The only operations that are carried out on the counter values are the increments c_i++, the resets $c_i := 0$ and $c_i := n_i$, and the comparisons $c_i = n_i$? and $c_i \neq n_i$?.

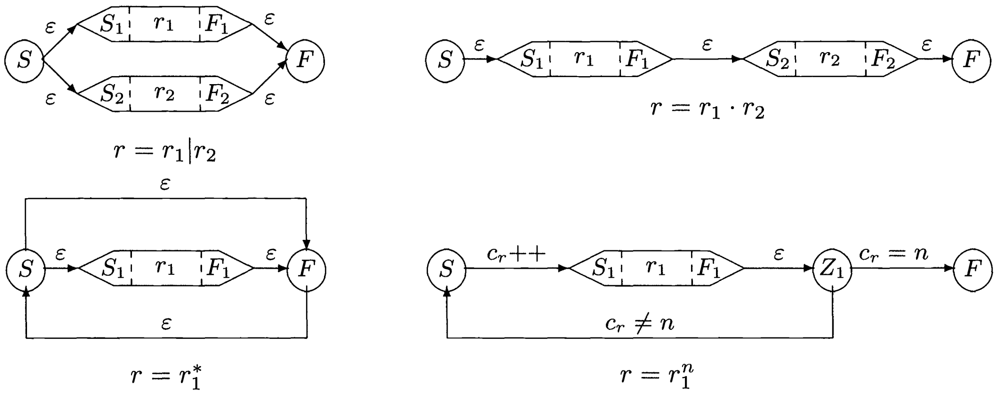
This automaton can be described by a transition graph whose nodes correspond to the states of Z and each of whose edges are labeled with an input symbol (possibly ε) or with input ε and one of the allowed counter actions. The transition graph can be obtained from a given regular expression r by recursively applying the rules shown in Figure 11. The resulting graph representing $\mathcal{A}$ has a description of size polynomial in the length of r.

Additionally, we require that $\mathcal{A}$ has the following properties:

(i) There exists a special final state F_{rev} which has a transition to each state of $\mathcal{A}$ for each input symbol (including ε) but which cannot be reached from any other state.

(ii) There are ε-transitions from each state of $\mathcal{A}$ to itself.

(iii) There exists a bound $c(r)$ that is polynomial in the size of r such that the following holds: if there exists a sequence of ε-transitions from a state Z_1 to a state Z_2, then Z_2 can be reached from Z_1 by $\leq c(r)$ ε-transitions.



$r = r_1|r_2$

$r = r_1 \cdot r_2$

$r = r_1^*$

$r = r_1^n$

FIG. 11. *A recursive construction technique for transition graphs of automata with counters. For each occurrence of an exponent in the underlying regular expression, a new counter must be introduced. Initially, all counters are set to* 0. *After a successful test $c_r = n$, the counter c_r has to be reset to* 0. (*This is omitted in the figure.*)

To obtain property (iii), we may render the construction of the automaton for $r = r_1^n$ in Figure 11 as follows: if $\varepsilon \in L(r_1)$ (which can be decided in polynomial time), then we add an ε-transition from S to Z_1, during which the counter c_r is set to n. This allows to "exit" an ε-loop by a constant (i.e., independent on n) number of transitions. (This property holds for the other construction elements by default.)

Step 2: *Construction of a scene $S_{\mathcal{A}}$ that simulates $\mathcal{A}$.* By using additional inputs to code the possible nondeterministic choices of $\mathcal{A}$, it is possible to build a combinational logic circuit with $O(\log(|Z| \cdot n_1 \cdots n_m))$ inputs and outputs which "evaluates" the transition diagram of $\mathcal{A}$ and whose size is polynomial in the size of $\mathcal{A}$. This circuit can be extended to a sequential logic circuit with binary registers for storing the state $(z, c_1, \ldots, c_m)$ of $\mathcal{A}$ and with external inputs for the next input symbol $x \in X \cup \{\varepsilon\}$ and the actual choice such that $\mathcal{A}$ is simulated step by step according to the chosen inputs when the circuit is clocked.

Using the constructions of the previous section, we can transform this circuit to a kinematic scene $S_{\mathcal{A}}$ which is able to simulate the work of $\mathcal{A}$ if a specific sequence of DH, COPY, DH′, and CLOCK is provided.

Nondeterminism can be captured by imposing no restriction on the value of the input that codes the choice of $\mathcal{A}$. We provide this input with a chain of binary memory cells to store its value such that it can be adjusted arbitrarily when the adequate DH signal is set to 0.

Step 3: *Construction of a scene $S_{\mathcal{B}}$ that produces arbitrary inputs for $S_{\mathcal{A}}$.* The scene $S_{\mathcal{B}}$ is assembled in three parts:

(i) a chain of binary memory cells for storing the value of the actual input x;

(ii) a mechanism SIGSEQ, similar to the mechanism in Figure 7, which produces a cyclical signal sequence that forces $S_{\mathcal{A}}$ to simulate the working of $\mathcal{A}$;

(iii) a mechanism TEST which can be forced to test at any time if the actual state z of $S_{\mathcal{A}}$ is a final state and which allows $S_{\mathcal{A}}$ to return to its initial configuration after this test process.

In order to simulate the work of $\mathcal{A}$, we have to ensure that $S_{\mathcal{A}}$ can make "arbitrary" ε-transitions between two "real" inputs generated by $S_{\mathcal{B}}$. To achieve this, SIGSEQ produces a sequence of $c(r) + 1 + c(r)$ repetitions of the signal sequence that is necessary to simulate a single transition. For the first $c(r)$ transitions, the input x is forced to code ε. For the next transition, the input x may be arbitrary (i.e., the value of x is not restricted). For the final $c(r)$ transitions, the input x is again forced to code ε.

In order to test if z is a final state, we simultaneously have to test if z contains an actual state (i.e., the test can only be performed if we are at the end of a clock sequence with DH = 1). If z is an actual state but is not final, then there will be a collision during the test. In the last phase of the test, S_A is allowed to return to its initial configuration, e.g., by "decoupling" the DH signal from SIGSEQ with the help of a copy mechanism.

The scene S_B can produce any input $w = x_1 \cdots x_n$ of arbitrary length and can simultaneously force S_A by the generated signal sequence to simulate some computation of the nondeterministic automaton $\mathcal{A}$ on w. By the TEST mechanism, we can ask if the simulated computation accepts w. If $L(r) = X^*$, then, given any motion of S_B containing a test, there always exists a motion of S_A such that S_A does not collide with S_B during the test. (S_A has to simulate an accepting computation of $\mathcal{A}$ for the input word generated by S_B before the test is performed.)

On the other hand, if there is a word $w \notin L(r)$ and we force S_A to simulate $\mathcal{A}$, then we get a collision by producing w as input and then performing the test, regardless of which nondeterministic computation is chosen by S_A.

We also have to take care of motions of S_B which do not generate a valid computation of the automaton $\mathcal{A}$. The only way to produce such "irregular" motions is by reversing the clock sequence. By the construction of the automaton, however, we can always get to the final state F_{rev} by a reversed transition and then stay there. Thus S_A can avoid a collision during a test after any irregular motion of S_B.

Finally, S_A can avoid a collision with S_B for any given (i.e., predefined) motion of S_B if and only if $L_{\text{rex}} = X^*$. This completes the proof. □

Since the TEST mechanism resets scene S_A to an initial state after testing, this proof also shows that the following problem is EXPSPACE-hard: decide if S_A can avoid the collision with one recursively predefined infinite motion of S_B.

The second problem that we investigate in this section concerns quantified queries on allowed configurations and can be stated as follows.

PROBLEM 2 (quantified query problem (QQP)). *Given a kinematic scene S with configuration space* $\mathrm{CS}(S) \subseteq \mathbf{R}^k$*, initial configuration* $C_0 \in \mathrm{FS}(S)$*, and quantifiers* $Q_i \in \{\exists, \forall\}$, $1 \le i \le k$, *decide if the formula*

$$Q_1 x_1 \in [0,1] \cdots Q_k x_k \in [0,1] : \ (x_1, \ldots, x_k) \in \mathrm{FS}_{C_0}(S) \tag{1}$$

is true.

This problem is similar to the EXPTIME-hard problem of deciding if a given formula in the theory of the real numbers $(\mathbf{R}, +, *, =, <, 0, 1)$ is true. However, in the QQP, we are restricted to formulas that describe the free space of some kinematic scene. Further, we require that the given formula must be satisfied by values that belong to a specific connected component of free space.

Our goal is to obtain a reduction of the decision problem in the theory of the real numbers to the QQP by constructing for any given formula a scene that evaluates this formula. By the planar mechanisms introduced in the previous section, we cannot realize the arithmetic operation of multiplication, but the decision problem in the theory of the reals $(\mathbf{R}, +, =, <, 0, 1)$ without multiplication is already known to be EXPTIME-hard [17]. We thus base our considerations on this theory and denote the

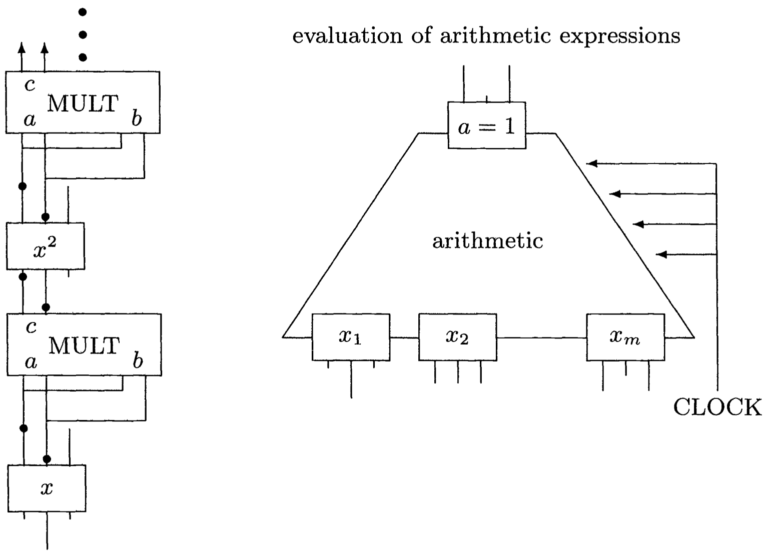


FIG. 12. *A scene for evaluating expressions in the theory of real numbers. The method of generating small numbers by squaring is illustrated on the left.*

corresponding satisfiability problem of quantified formulas as TH(**R**,+). To model this problem in the context of kinematic scenes, we have to solve the following: in our constructions, we can represent only real numbers in the interval $[0, 1]$, or at least in a bounded interval, since all objects have a bounded size. Here we can use the fact that there exists a general decision algorithm for the satisfiability problem considered which is based on the fact that it is sufficient to test the correctness of the formula for a finite subset of some domain $D = [-2^{2^{cn}}, 2^{2^{cn}}]$, where n corresponds to the length of the formula and c denotes a fixed constant.

Thus we can confine our investigations to a bounded domain, but there still remains the problem of how to code this domain. It is not possible to use mechanisms of size $2^{2^{cn}}$ since this would lead to scenes with descriptions of exponential size. We shall code the domain D by compressing it to $[-1/2, 1/2]$ and then shift this interval to $[0, 1]$. However, this leads to the problem of how to represent the numbers 0 and 1, i.e., the allowed coefficients in the input formulas, which then correspond to $\hat{0} = 1/2$ and $\hat{1} = 1/2 + 1/2 \cdot 1/2^{2^{cn}}$. To obtain the small offset $1/2^{2^{cn}}$, or, more generally, to get a coding of the large domain D, we have to use multiplications and thus planar linkages (i.e., arbitrary scenes with rotational degrees of freedom). We build a mechanism that squares a given number $x \in [0, 1]$ cn times, specified in a configuration that corresponds to $x = 1$ (i.e., with a polynomial description). Then we have access to the number $1/2^{2^{cn}}$ in the context of a motion-planning problem by requiring x to move to a position that codes 1/2 (see Figure 12). By multiplying the result with 1/2 and then adding 1/2, we can construct a mechanism that generates the number $\hat{1} = 1/2 + 1/2 \cdot 1/2^{2^{cn}}$ if some clock signal is set to 1.

Because of the necessity of multiplications to code the domain D, we can prove the following theorem only for arbitrary kinematic scenes with rotational degrees of freedom.

THEOREM 3.2. *The QQP is EXPTIME-hard.*

Proof (TH($\mathbf{R}$,+) $\leq$ QQP). For any given formula F' of length n in the theory of $(\mathbf{R}, +, =, <, 0, 1)$, we construct a kinematic scene S with an initial configuration C_0 such that formula (1) holds if and only if the given formula is true.

Without loss of generality, we can assume that formula F' is given in prenex normal form $F' = Q'_1x'_1 \cdots Q'_mx'_m : E'(x'_1, \cdots, x'_m)$, where E' denotes a Boolean expression with linear inequalities as atoms.

By the tools presented in the previous section, it is easy to construct a scene S with m variable cells that code $\hat{x}'_1 = (1/2^{1+2^{cn}})x'_1 + 1/2, \dots, \hat{x}'_m = (1/2^{1+2^{cn}})x'_m + 1/2$ which evaluates E' to a logical variable a if a CLOCK signal is set to 1 (see Figure 12).

The atomic inequalities in E' can be written as additions and subtractions of variables and the number 1, followed by a comparison with 0. Thus we need the value $\hat{1}$ only as a constant which we can add to or substract from a variable. For each occurrence of 1 in E', scene S contains a mechanism which generates $\hat{1}$ as adequate input if the CLOCK signal is set to 1. Since we are interested in values that satisfy the expression E', we fix the evaluation result a in the position that codes the logical value 1.

Scene S and some initial configuration C_0 with CLOCK $= 0$ can be constructed in polynomial time. We assume (a) that the parameters $x_1, \dots, x_k$ that span the configuration space begin with the positions of the rods used to code $\hat{x}'_1, \dots, \hat{x}'_m$ and (b) that the last parameter x_k corresponds to the value of the CLOCK. The quantifiers $Q_1, \dots, Q_m$ in formula (1) are defined by $Q_1 = Q'_1, \dots, Q_m = Q'_m$, $Q_{m+1} = \exists, \dots, Q_{k-1} = \exists$, $Q_k = \forall$.

We define $E(x_1, \dots, x_k)$ to be the formula $Q_{m+1}x_{m+1} \dots Q_kx_k : (x_1, \dots, x_k) \in \mathrm{FS}_{C_0}(S)$. Then (1) can be written as $Q_1x_1 \dots Q_mx_m : E(x_1, \dots, x_k)$.

Assume the expression E' is true for specific values of $x'_1, \dots, x'_m$. Since E' can be evaluated to 1, there exists a motion in S starting at the initial configuration during which first the objects that code $\hat{x}_1, \dots, \hat{x}_m$ are set to the corresponding values and then all other objects are positioned such that the CLOCK can switch from 0 to 1. Thus the expression E is also true for the corresponding values (i.e., the same values respective to the linear transformation of the domain).

If, on the other hand, E is true for specific values of $x_1, \dots, x_m$, then there exists a motion in S during which E' is evaluated to 1. Thus E' is also true for the corresponding values.

Since all values in $[0, 1]$ can be adjusted in S respective to the given initial configuration, we get that (1) is true if and only if F' is true. (Recall that the satisfiability of F' is determined by its satisfiability in D.) This completes the reduction. □

We remark that the CAP and QQP are both problems that concern classes of motions or configurations. Exponential lower bounds for problems that concern the existence of single motions seem to be impossible to prove within the framework of kinematic scenes. Coding computations of exponentially time-bounded Turing machines by motions requires the possibility of distinguishing between double exponentially many configurations. However, this is impossible in a free space of roughly single exponential combinatorial complexity.

In the next section, we investigate extended models of scenes that involve physical constraints, in which we have the possibility of procesing more information than in kinematic scenes, and return to the classical existence problem of motions.

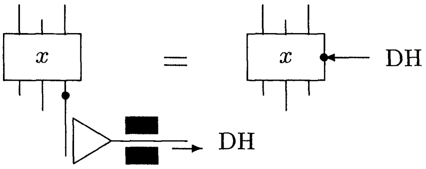


FIG. 13. *A memory cell in the dynamic model.*

4. Motion planning under physical constraints.

4.1. Undecidability results. We first investigate motion planning in the model of dynamic scenes for the idealized case where motions may be forbidden (i.e., have infinite costs) if specific sides of two objects are sliding along each other tangentially during this motion. According to the model of dynamic scenes introduced, we assume that this effect occurs due to friction and code it in the input by marking adequate sides of the objects with the symbol "∞". We consider tangential movements of two objects along touching sides as illegal if both sides concerned are marked with "∞".

The essential difference between this model and the model of kinematic scenes is that we now have the possibility of fixing objects in arbitrarily many positions and thus storing an infinite amount of information in the position of one object. Figure 13 shows a memory cell that is able to store any real number $x \in [0, 1]$ by setting the value of a DH signal to 1. The storage state is established by "pressing" a side of a triangle against the rod of a variable cell x, where the position of this triangle corresponds to the value of DH. (The touching sides of the triangle and of the rod are marked with "∞").

Since the constraints considered concern only the existence of motions, the allowed configurations of a dynamic scene are the same as the allowed configurations of the underlying kinematic scene. Thus motion planning in this model cannot be expressed as a simple path search in some semialgebraic subspace of the Euclidean space. In fact, the dynamic motion-planning problem as stated below is undecidable since it opens the possibility of processing an infinite amount of information and thus simulating arbitrary Turing machines with infinite tape.

PROBLEM 3 (dynamic motion-planning problem (DMPP)). *Given a dynamic scene S with forbidden movements and initial and final configurations C_i and C_f, decide if there exists an allowed motion that connects these configurations.*

THEOREM 4.1. *The DMPP is undecidable.*

Proof (simulation of arbitrary Turing machines). We show that we can construct a dynamic scene S for any given, deterministic Turing machine M and an initial configuration $C_i \in \mathrm{FS}(S)$ for any input w of M such that there exists a motion in S connecting C_i with some final configuration C_f if and only if M accepts w.

We assume that M has only one tape and an alphabet with p symbols, including the blank. Obviously, M can be simulated by a push-down automaton with two stacks, one that corresponds to the part of the tape to the left of the actual position of the head and one that corresponds to the part below and to the right of the head. The actions of M, especially movements of the head, can be expressed by stack operations. We interpret the contents of these stacks as p-adic numbers $x = 0.x_1x_2x_3\ldots$ and $y = 0.y_1y_2y_3\ldots$ in $[0, 1]$, as shown in Figure 14, stored in adequate memory cells. The actions of the Turing machine M can now be realized by the logical and arithmetic operations that we introduced in §2.

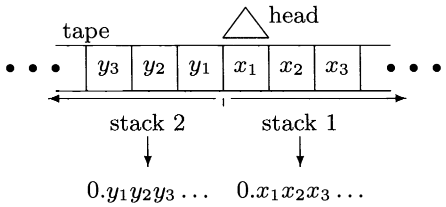


FIG. 14. *Coding of the tape of a Turing machine by real numbers.*

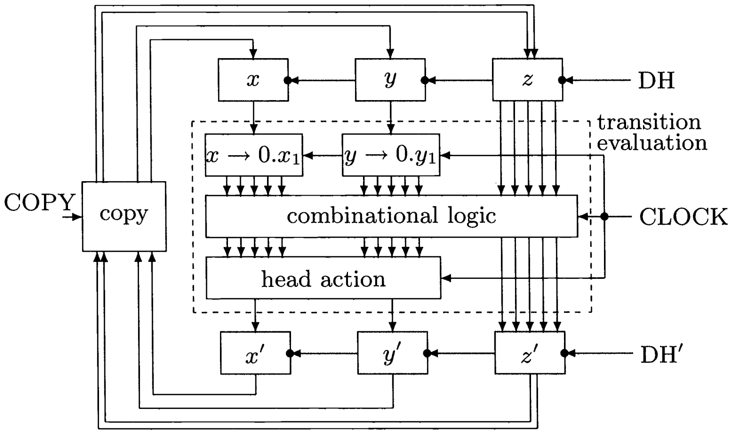


FIG. 15. *Assembly of a scene for simulating arbitrary Turing machines. The cells labeled with x, y, x', and y' denote memory cells for real values; z and z' can be stored in chains of binary memory cells.*

The transition table of M with binary coded inputs and outputs can be "evaluated" by a combinational logic circuit and thus by a kinematic scene. Since the actual state can be stored in a chain of binary memory cells, it remains only to obtain a binary representation of the actual input symbol x_1 from x, to write a new symbol x_1' given by a binary representation to x, and then to simulate the movement of the head. In order to get access to x_1, we can use our comparison mechanism to simultaneously test for each integer $0 \leq i \leq p-1$ if $i/p \leq x < (i+1)/p$ and calculate a binary representation of x_1 from the results. In the opposite direction, the construction of the real number $0.x_1'$ from a binary representation of x_1' can trivially be attained by additions and multiplications with constants. The substitution of x_1 by x_1' is done by adding $0.x_1' - 0.x_1$ to x. To realize a movement of the head to the right, we have to "pop" x_1' from x and to "push" it to y. This translates to $x := (x - 0.x_1') \cdot p$ and $y := ((1/p) \cdot y) + 0.x_1'$. Movements to the left also require analysis of y by comparisons in order to calculate a logical representation of y_1.

The simulation principle is shown in Figure 15. We need three memory elements x, y, and z for the actual configuration of M and three elements x', y', and z' to store the transition result until it is copied to the actual configuration. The whole mechanism is controlled by the signals DH, COPY, DH′, and CLOCK, which are generated analogously to the case of the binary counter (see Figures 6 and 7).

We remark that motions in the constructed scene need not correspond to regular computations of M since transitions of M can also be carried out in reverse order. Nevertheless, there exists a motion from a given initial configuration that corresponds

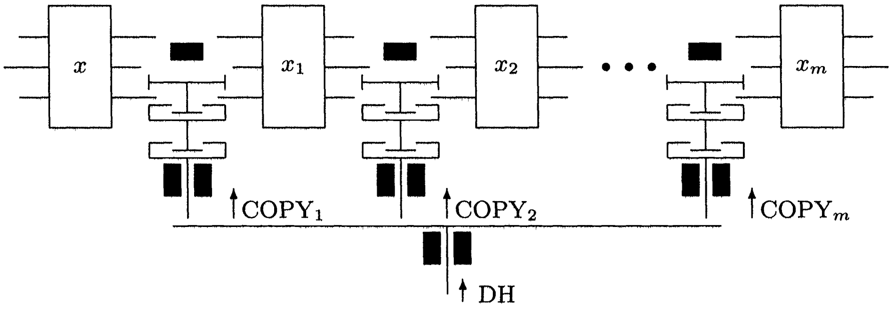


FIG. 16. *A memory cell for storing the real value x in the context of passive motion planning.*

to an input w to some final configuration that corresponds to the (without loss of generality, unique) accepting state of M if and only if M accepts w. □

The essential fact that makes this proof work is that we can fix objects in arbitrary positions, achieved by restricting the set of allowed motions. Dynamic scenes are a powerful tool for defining such restrictions, and there are other practical problems in which we have weaker possibilities of defining restrictions on motions but in which we nevertheless get a similar situation. We introduce one such problem, which we call *passive motion planning*, and show that this problem is also undecidable.

In the standard formulation of motion planning, we assume that the objects can move independently on their own, as if by a magic hand. However, in practical applications, the objects often have to be moved by an external robot system that is able to move only a bounded number of objects at the same time. We thus allow only motions during which not more than a given number m of objects are moved simultaneously. Further idealized by considering only translational movements in one direction and translated into the formalism of configuration space, we define passive motion planning as follows.

PROBLEM 4 (passive motion-planning problem (PMPP)). *Given a kinematic scene S, initial and final configurations C_i and C_f, and an integer m, decide if there exists a path in* $\text{FS}(S) \subseteq \mathbf{R}^k$ *that connects C_i with C_f which consists of a finite number of straight line segments, where each segment is orthogonal to at least $k - m$ of the k coordinate axes of $\mathbf{R}^k$.*

THEOREM 4.2. *The PMPP is undecidable.*

Proof. Passive motion planning allows us to fix the contents of a variable cell according to a DH signal by connecting the cell to m other variable cells via m copy mechanisms (see Figure 16). Since we must be able to move at most m objects at the same time to establish the connection, we force all copy mechanisms to get to their connecting position by setting a DH signal to 1, but we do not rigidly connect them to this signal.

Given an arbitrary Turing machine M and an input w, we first construct a dynamic scene S' that is able to simulate M, as in the proof of Theorem 4.1. Let m' be the number of movable objects in S'. We substitute the memory cells for real-valued variables by the mechanisms described above with $m = m'$. The input for the PMPP consists of the resulting kinematic scene S, an initial configuration C_i that corresponds to w, a final configuration C_f that corresponds to the accepting state of M, and the bound $m = m'$.

Since any two configurations in $\text{FS}_{C_i}(S')$ that can be connected by some finite motion can also be connected by a motion that consists of a finite sequence of translations of the m movable objects, the restrictions that are imposed on allowed motions by passive motion planning only affect the memory cells in S. These are forced to store information by setting the DH signals to 1. Thus the allowed motions in S and S' are essentially the same. □

In this subsection, we have investigated motion-planning problems involving constraints by which motions could be forbidden. In the next subsection, we weaken this restriction by assigning costs to motions such that we only have the possibility of making some motions more expensive than others.

4.2. Exponential lower bounds. In this subsection, we investigate arbitrary dynamic scenes in which finite costs are implicitly defined for motions by assigning integers, denoted as friction coefficients, to the sides of the objects. We do not need to specify a sound physical model of how to define friction. (In such a model, we would also have to deal with forces or masses.) We shall need only the fact that some motions can be made expensive in comparison to others. The goal in dynamic motion planning is to find a motion which stays under a given cost limit.

If we allow that some motions cost nothing (i.e., consume no energy), we have a situation that is symmetric to the idealization that some motions have infinite costs. To translate a given problem that involves infinite cost to an equivalent problem that involves cost 0, we only have to assign to all motions with infinite cost (i.e., forbidden motions) some finite cost and to all other motions cost 0. The existence of an allowed motion then corresponds to the existence of a motion with cost 0, making it possible to carry the undecidability results of the previous subsection over to this problem.

We thus require that during any motion, some sides tangentially slide along each other. For the case of planar scenes, this can be achieved by assuming that all objects are sliding on some ground floor. For the sake of completeness, we define the cost of a path (resp., motion) as follows: if two sides with friction coefficients a and b slide along each other during parts of the motion and if l denotes the Euclidean length of the relative movement of the two objects involved, then this pair of sides causes the cost $a \cdot b \cdot l$. The cost of the entire path results from summarizing these costs.

PROBLEM 5 (cheapest motion-planning problem (CMPP)). *Given a dynamic scene S with friction coefficients $\in \mathbf{N}$ such that any motion of positive length has some cost > 0, initial and final configurations C_i and C_f, and a cost limit $m \in \mathbf{N}$, decide if there exists a motion with cost $< m$ that connects these configurations.*

THEOREM 4.3. *The CMPP is EXPTIME-hard if the friction coefficients are coded by mantissa and exponent.*

Proof. Based on the construction introduced in the proof of Theorem 4.1, we build dynamic scenes that are able to simulate exponentially time-bounded Turing machines in the context of cheapest motion planning. Since we cannot fix objects like in this proof, we lose the capability of storing arbitrary information. However we still keep the possibility of coding a finite amount of information which depends not only on the underlying kinematic scene but also on the friction coefficients and the cost limit m.

Assume that we are given a $c2^n$ time-bounded, deterministic Turing machine M and an input string w of length n. Let S' be the dynamic scene constructed for M in the proof of Theorem 4.1, and let C_i and C_f be the initial and final configurations that correspond to the initial state of M with input w and the accepting state of M.

We first assign the friction coefficient 1 to all object sides of S' with the exception of those sides that are marked with "∞". Then, if there exists an allowed motion in S'

that connects C_i and C_f, there also exists an adequate motion in the resulting scene with cost $< c'c2^n$, where c' is a constant only depending on S' (or M, respectively). We set the cost limit to $m = c'c2^n$ and define S by substituting the "∞" marks by the friction coefficient $c_f = mp^{2c2^n}$, where p denotes the size of the alphabet.

The cost limit has been chosen such that if M accepts w, there exists a motion in S that connects C_i and C_f with cost $< m$. It remains to show that there exists no such motion if M rejects w. In this case, there exists no allowed motion in S' that connects C_i with C_f, though the underlying kinematic scenes are the same. Thus a connecting motion in S must include some movements which are forbidden in S', i.e., movements of the rods of the memory cells that code the tape while their DH signal is 1. However, these movements are expensive in S, and in order to stay under the given limit m, such a movement cannot change the position of a rod by a distance of more than $m/c_f^2 < 1/p^{2c2^n}$. Hence the only "actions" which can be "performed" by a connecting motion in S, in addition to what is possible by allowed motions in S', correspond to changes of the tape at positions, which are more than $2c2^n$ tape cells away from the actual head position. These positions can never be reached during a computation of M that is started with input w. (M uses at most $c2^n$ cells of the tape!) Since the possible changes in the contents of a memory cell during a storage state are so small that they do not affect the simulation process, we can translate a motion in S that connects C_i and C_f with cost $< m$ to an accepting computation of M (or, more strictly, of the symmetric Turing machine corresponding to M)—a contradiction to the presupposition that M rejects w.

The size of the underlying kinematic scene does not depend on n. C_i, C_f, and the cost limit m have a binary description of size polynomial in n, and the friction coefficients (especially c_f) can be coded in a length polynomial in n by a number representation using mantissa and exponent. The input of the CMPP can be computed for a given M and w in time polynomial in n. This completes the proof. □

In the proofs of Theorems 4.1–4.3, we used planar scenes consisting of polygonal objects with purely translational freedom of motion. Thus the motion-planning problems considered correspond to path-search problems in multidimensional polyhedra, specified by Boolean expressions of linear inequalities of size polynomial in the size of the scenes.

The CMPP can be interpreted as the problem of finding a path with cost $< m$ in a polyhedron in which specific cost functions are associated with the sides. We shall focus on this aspect in the next subsection.

4.3. A geometrical formulation of Turing computability. In the proof of Theorem 4.1, we introduced a technique for simulating Turing machines by dynamic motion-planning problems. Translated into the formalism of configuration space, our constructions also provide us with an interesting geometrical interpretation of the notion of Turing computability.

We again consider dynamic scenes with forbidden motions, defined by marking some sides in a kinematic scene with the friction coefficient ∞. To capture the restrictions on motions in dynamic motion planning, we define the *free space of a dynamic scene* as the free space $\mathrm{FS}(S) \subseteq \mathbf{R}^k$ of the underlying kinematic scene S supplemented by a function $f : \mathrm{FS}(S) \to \mathrm{Power}(\mathbf{R}^k)$, which assigns to each point $p \in \mathrm{FS}(S)$ a set of vectors such that a path $w : [0,1] \to \mathrm{FS}(S)$ corresponds to an allowed motion in the dynamic scene if and only if the tangent vector at any point $p \in w([0,1])$ is in the set $f(p)$. Since f defines a structure on $\mathrm{FS}(S)$ which no legal path is allowed to "leave," we shall further denote the tuple $(\mathrm{FS}(S), f)$ as a *structured space.*

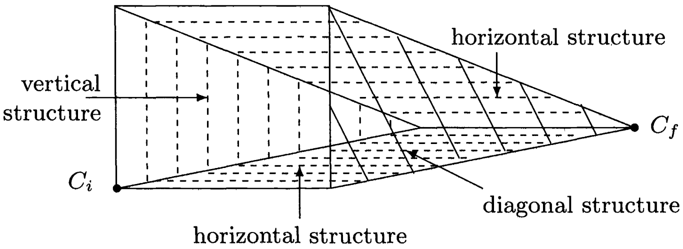


FIG. 17. *A structured polyhedron. The polyhedron consists of four structured sides. For example, the triangle on the right can be crossed only in the diagonal direction.*

This notion provides us with a powerful tool for describing physical problems. By allowing only vectors of specific lengths or directions as tangents of allowed paths, problems involving constraints on object velocities or path curvature can be represented. To plan "smooth motions" of a rectangular object (e.g., a car) with translational and rotational freedom in some given planar environment, the allowed tangent vectors at each point $p = (x, y, \theta)$ of free space may be defined as $f(p) = \{(\alpha \sin(\theta), \alpha \cos(\theta), \theta'); \ \alpha \in [a, b], \ \theta' \in [-\delta, +\delta]\}$ with given constants a, b, and δ.

Compared to the general model, the structured spaces that occured in this section constitute a very restricted subclass. The free space of a kinematic scene is a semialgebraic space which can be decomposed into disjoint, open, semialgebraic cells, where each cell is homeomorphic to the Euclidean space $\mathbf{R}^m$ for some $m \in \mathbf{N}_0$ [1]. In the special case of polygonal objects with purely translational freedom of motion, the free space is a polyhedron and the cells can be triangulated into simplices. The function f which defines the structure of the free space of the scenes constructed in the proof of Theorem 4.1 is constant on each open simplex s of the triangulation, i.e., $f(p) = f(p') \ \forall p, p' \in s$. Since the constraints force tangent vectors to be zero only in some coordinates, the value of f corresponds to some linear subspace, or hyperplane, of the Euclidean space $\mathbf{R}^k$. We refer to *linearly structured polyhedra* if the set of allowed vectors $f(p)$ for $p \in s$, where s is an open simplex of the polyhedron, is defined as the space spanned by a given set of vectors.

Since the dynamic scenes constructed in the proof of Theorem 4.1 for given Turing machines and input strings depend only on the transition functions, we can interpret Turing machines as linearly structured polyhedra. The points of a linearly structured polyhedron can thus be compared with machine configurations and the allowed paths with computations.

Figure 17 provides us with some intuition about structured polyhedra. There are situations such that there exist paths starting at a given initial configuration which come arbitrarily close to a given final configuration, though there is no allowed path of finite length connecting these configurations. The situation described corresponds to a Turing machine which interprets the contents of its tape as the real number 1, successively divides this number by 2, and thus approaches but never reaches 0.

We conclude with the following theorem, which provides an alternative, geometrical definition of Turing computability.

THEOREM 4.4. *A language $L \subseteq \{0, 1\}^*$ is Turing computable if and only if there exist a linearly structured polyhedron P, an edge $e \subseteq P$, a linear function h that* *embeds* $\{0,1\}^*$ *as binary number* $0.w$ *in* e, *and a point* $q \in P$ *such that*

$$w \in L \iff \exists \text{ allowed path from } h(w) \text{ to } q \text{ in } P. \tag{2}$$

Proof. The construction of dynamic scenes for arbitrary Turing machines shows that a structured polyhedron P (together with e, h, and q) that satisfies condition (2) exists if L is Turing computable.

The opposite direction—i.e., for any given P, e, h, and q, the language L that satisfies (2) is Turing computable—follows from the fact that we can enumerate all points of P that are reachable from a given point $\in e$. (The algorithm has to follow the structure of P inductively, simplex for simplex, by marking the already detected reachable areas). □

5. Future work. We think that the ideas which we developed in this paper are interesting from both a complexity-theoretic and a physical point of view.

Because of the intuitive power of geometrical reasoning and also because of the close relationship to physical problems, it seems to be a challenging subject of research to develop a complexity theory based on the geometrical machine model introduced in Theorem 4.4.

Interesting questions in this context are how known characteristics of Turing machines (number of tapes, nondeterminism, etc.) are reflected in the geometrical model and how characteristics of the geometrical model could help to classify problems in complexity theory. We state the following concrete problems:

1. The number of tapes of a Turing machine corresponds to the number of different structures on the simplices of P. Which restrictions on the possible structures lead to interesting subclasses of the class of Turing-computable languages?
2. Can two structured polyhedra that accept the same language be transformed to each other by some reasonable set of given transformation rules?
3. The dimension of the polyhedron that we get by taking the roundabout way via dynamic scenes depends on the Turing machine M. Is it possible to find for each structured polyhedron an "equivalent" three-dimensional polyhedron?

The latter of these problems is related to probably the most interesting question in practice, namely, to detect the complexity of motion planning under constraints for lower-dimensional problem instances, especially instances involving only one movable object.

Acknowledgments. The author would like to thank Günter Hotz for his support, helpful discussions, and suggestions.

REFERENCES

[1] J. T. Schwartz and M. Sharir, *On the piano movers problem* II: *General techniques for computing topological properties of real algebraic manifolds*, Adv. Appl. Math., 4 (1983), pp. 298–351.

[2] C.-K. Yap, *Algorithmic Motion Planning*, J. T. Schwartz and C.-K. Yap, eds., Algorithmic and Geometric Aspects of Robotics 1, Lawrence Erlbaum Associates, Hillsdale, NJ, 1987, pp. 95–143.

[3] J. Canny, *A new algebraic method for Robot motion planning and real geometry*, in Proc. 28th IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1987, pp. 39-48.

[4] ———, *Some algebraic and geometric computations in PSPACE*, in Proc. 20th ACM Symposium on the Theory of Computing, Association for Computing Machinery, New York, 1988, pp. 460–467.

[5] M. Sharir, *Algorithmic motion planning in robotics*, Technical Report 392, New York University, New York, 1988.

[6] J. Reif, *Complexity of the mover's problem and generalizations*, in Proc. 20th IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1979, pp. 421–427.

[7] J. E. Hopcroft, D. A. Joseph, and S. H. Whitesides, *Movement problems for 2-dimensional linkages*, SIAM J. Comput., 13 (1984), pp. 610–629.

[8] J. E. Hopcroft, J. T. Schwartz, and M. Sharir, *On the complexity of motion planning for multiple independant objects: PSPACE-hardness of the warehouseman's problem*, Internat. J. Robotics Res., 3 (1984), pp. 76–88.

[9] P. Spirakis and C.-K. Yap, *Strong NP-hardness of moving many discs*, Inform. Process. Lett., 19 (1984), pp. 55–59.

[10] S. Fortune and G. Wilfong, *Planning constrained motion*, in Proc. 20th ACM Symposium on the Theory of Computing, Association for Computing Machinery, New York, 1988, pp. 445–459.

[11] P. Jacobs and J. Canny, *Planning Smooth Paths for Mobile Robots*, in Proc. IEEE International Conference on Robotics and Automation, IEEE Computer Society Press, Los Alamitos, CA, 1989, pp. 2–7.

[12] J. Canny, A. Rege, and J. Reif, *An exact algorithm for kinodynamic planning in the plane*, in Proc. 6th Annual ACM Symposium on Computational Geometry, Association for Computing Machinery, New York, 1990, pp. 271–280.

[13] J. Canny and J. Reif, *New lower bound techniques for robot motion planning*, 28th IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1988, pp. 306–318.

[14] J.-C. Latombe, *Robot Motion Planning*, Kluwer Academic Publishers, Norwell, MA, 1991.

[15] H. R. Lewis and C. H. Papadimitriou, *Symmetric space-bounded computation*, Theoret. Comput. Sci., 19 (1982), pp. 161–187.

[16] A. B. Kempe, *On a general method of describing plane curves of the nth degree by linkwork*, Proc. London Math. Soc., 7 (1876), pp. 213–216.

[17] J. E. Hopcroft and D. Ullman, *Introduction to Automata Theory, Languages and Computation*, Addison–Wesley, Reading, MA, 1979.

SIAM J. COMPUT.
Vol. 25, No. 6, pp. 1254–1267, December 1996

006

AVERAGE AND RANDOMIZED COMPLEXITY OF DISTRIBUTED PROBLEMS*

NECHAMA ALLENBERG-NAVONY†, ALON ITAI‡, AND SHLOMO MORAN‡

Abstract. Yao proved that in the decision-tree model, the average complexity of the best deterministic algorithm is a lower bound on the complexity of randomized algorithms that solve the same problem. Here it is shown that a similar result does not always hold in the common model of distributed computation, the model in which all the processors run the same program (which may depend on the processors' input).

We therefore construct a new technique that together with Yao's method enables us to show that in many cases, a similar relationship does hold in the distributed model. This relationship enables us to carry over known lower bounds on the complexity of deterministic computations to the realm of randomized computations, thus obtaining new results.

The new technique can also be used for obtaining results concerning algorithms with bounded error.

Key words. distributed computing, randomized algorithms, average complexity, message complexity, bit complexity, lower bounds, Yao's lemma

AMS subject classifications. 68Q22, 68Q25

1. Introduction.

In 1977, Yao presented results relating the average deterministic complexity and the randomized complexity of the same problem in the decision-tree model [9]. In particular, he introduced "Yao's inequality", which states that the average complexity of the best deterministic algorithm is a lower bound on the complexity of randomized algorithms that solve the same problem. As Yao pointed out, this inequality may be applied to derive lower bounds on the randomized complexity from known lower bounds on the average complexity.

Yao's lemma can be immediately applied to additional computational models. For example, the parallel random-access machine (PRAM) model (see [5]). However, the following example shows that Yao's technique cannot be applied directly to the common distributed model.

The counterexample. Consider computing the AND function on an asynchronous ring. Every processor has its own private bit $x_i \in \{0,1\}$. Every deterministic algorithm for this problem has bit complexity $\Omega(n^2)$ [2]. Moreover, Attiya et al. showed that the worse case occurs for the input $\vec{1} = (1,1,\ldots,1)$ (i.e., every algorithm that is correct for all inputs $(x_1,\ldots,x_n) \in \{0,1\}^n$ requires $\Omega(n^2)$ communication bits for $\vec{1}$ under the same schedule S_0). Consider the distribution P:

$$P(\vec{x}) = \begin{cases} 1, & \vec{x} = \vec{1}, \\ 0 & \text{otherwise.} \end{cases}$$

Under this distribution, the worst case occurs with probability 1; hence the average number of communication bits is also $\Omega(n^2)$.

However, by using a randomized algorithm to choose a leader ($O(n \log n)$ bits [7]) and then have the leader send a message that computes the cumulative AND, the problem can be solved in $O(n \log n)$ bits by a randomized algorithm.

* Received by the editors July 22, 1993; accepted for publication (in revised form) February 14, 1995. Part of this work was conducted while the second and third authors visited AT&T Bell Laboratories, Murray Hill, NJ 07974.

† Computer Science Department, Hebrew University, Jerusalem 91904, Israel.

‡ Computer Science Department, Technion–Israel Institute of Technology, Haifa 32000, Israel.

Thus the upper bound on the complexity of randomized algorithms is strictly less than the lower bound on the average cost of deterministic algorithms. □

We cannot directly apply Yao's inequality for two reasons:

1. There is a basic (though somewhat implicit) assumption underlying Yao's inequality. This assumption is that randomized algorithms can be represented as a probability distribution over a set of deterministic algorithms. It turns out that this assumption depends on the model of computation studied, and we will see that this assumption does not hold for the common model of distributed algorithms, in which all the processors run the same program. Thus this technique cannot be used indiscriminately.

2. Even when the above assumption holds, we have to investigate the dependency on the schedule.

We consider a new technique that enables us to extend Yao's inequality to a very widely considered case of distributed models—the case in which each processor is guaranteed in advance to have a distinct private input (or, as is sometimes phrased in the literature, each processor is given a unique id).

This result is achieved in two steps. First, we "encapsulate" the relevant parts of Yao's technique by restating the lemma to meet our needs. Using this formulation, it is observed that Yao's inequality is not valid for the distributed model. Then we add a new technique to show that this inequality can be carried on to the model in which the processors have distinct ids.

These new results enable us to carry over several known lower bounds, from deterministic computations to randomized ones. Some of the lower bounds obtained by our technique are new, while others had been known before. However, we are able to extend the known lower bounds to more general settings, such as allowing algorithms that may make mistakes (with small probability).

Note that a lower bound on the restricted problem when the processors are assumed to have distinct ids also holds for the general problem. Thus the lower bounds we obtain are satisfied in the more general setup.

Like Yao's lemma and unlike most lower-bound proofs, our technique is independent of the topology of the network and holds for many complexity measures and different distributed models.

Yao has generalized his inequality to algorithms with bounded error. Using our technique, we carry this result to the distributed model and show that in some cases, the cost of distributed randomized algorithms with bounded error is bounded by the cost of error-free distributed deterministic algorithms.

Independently, Bodlaender [3] proved a result similar to our Corollary 3.3 and Theorem 4.1. However, there seems to be no direct way to extend his results to deal with randomized algorithms that can make errors (even when the error probability is 0). Thus the lower bounds obtained by our methods are stronger in the sense that they hold in more general settings.

2. Preliminaries.

2.1. Distributed systems.

A *distributed network* of size n consists of a strongly connected directed graph of n vertices. Each vertex corresponds to a processor, which is our basic computing unit.

Every edge of the graph represents a directed communication channel. These edges are the only means of communication between processors. We associate with each channel an unbounded first-in first-out (FIFO) queue of pending *messages.*

Each such processor has its own internal memory, program, program counter, in-buffer, and out-ports. The in-buffer of the processor—not to be confused with the queue of the edges—contains the messages that have arrived but have not yet been processed by the processor. Each out-port corresponds to a distinct outgoing edge. Since we are not concerned with computation time, we consider each processor as a (possibly infinite) state machine represented by its transition table. In other words, each configuration of the processor (memory content, program counter, step counter, and buffer state) will be represented by a different state. (Since the step counter strictly increases, a processor never returns to a previous state.)

Every step of the computation corresponds to a transition of the state of the processor. A single transition of a processor consists of receiving (zero or more) messages from some of its incoming channels (i.e., moving a message from the queue of pending messages of the incoming edge to the in-buffer), removing messages from its in-buffer, changing its state, and sending (zero or more) messages (i.e., placing messages on queues of its outgoing channels). The new state depends on the previous state and the message just received.

A *distributed algorithm* is the n-tuple of the state diagrams of the processors. The distributed algorithm is *uniform* if all the processors have the *same* state diagram. In this case, the processors are identical. We are interested in uniform distributed algorithms. (Since the processors may differ with respect to the incoming and outgoing degree of the corresponding nodes, we assume that all the processors have the same number of out-ports. However, for vertex v, only the first $\text{out\_deg}(v)$ ports correspond to edges of the network. Any attempt to write to an unassigned port results in an improper termination of the algorithm.)

Let X and Y be two sets, called the *private input set* and *private output set*, respectively. In our distributed model, a processor's actions may depend on its private input $x \in X$. In other words, each private input x corresponds to a different initial state. We also assume that each processor v_i has a write-once register, on which it writes its *private output*, y_i.

We require X to be a countable set. However, this restriction is not severe; it is implied when each private input can be represented by a finite number of bits. (There need not be a bound on the length of all the private inputs of the processors.)

The order by which the various processors are activated and the delays on the channels are governed by the *schedule.* The validity and efficiency of distributed algorithms often depend on the class of schedules allowed. We give below a definition of an *oblivious schedule class.* However, our technique is also valid for other schedule classes.

A *schedule* $S = (e_1, e_2, \ldots)$ is an infinite sequence of edges.

We now describe the ith step. Let $e_i = (u_i, v_i)$ be the ith component of S. If there are any pending messages in the queue of e_i, the first message is moved from the queue of pending messages of e_i to the in-buffer of v_i. Processor v_i is then *enabled*: it reads and removes some of the messages from its in-buffer and makes a state transition.

Example 2.1. A possible execution of the schedule $S = (e_1, e_3, e_4, e_5, e_5, \ldots)$ as applied to the following network. (See Figure 1 and Table 1.)

An *execution* is a sequence $\epsilon = (\epsilon_1, \epsilon_2, \ldots)$, where $\epsilon_i = (v, \text{IN}, \text{OUT}, s)$ such that v is the processor enabled at step i, IN is the set of messages received by v at that step, OUT is the set of messages v sent, and s is the new state of v.

Let us note that given a distributed deterministic algorithm A, an input $\vec{x} \in X^n$, and a schedule S, the execution is uniquely determined.

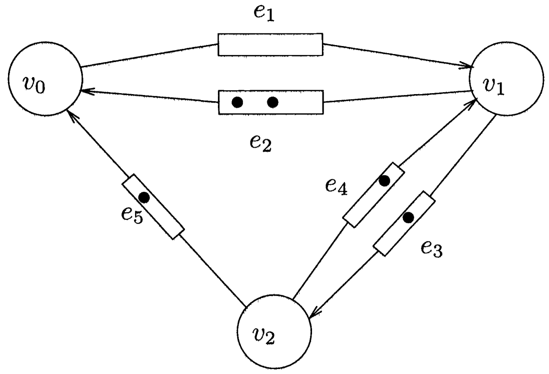


FIG. 1. *The network after executing the fourth step.*

TABLE 1

S	vertex	e_1	e_2	e_3	e_4	e_5	legend
		–	–	–	–	–	Initially, all buffers are empty.
e_1	v_1	–	m_1	m_2	–	–	v_1 sends messages m_1, m_2.
e_3	v_2	–	m_1	–	m_3	m_4	v_2 receives m_2, sends m_3, m_4.
e_4	v_1	–	m_1, m_5	–	–	m_4	v_1 receives m_3, sends m_5.
e_5	v_0	–	m_5	–	m_3	–	v_0 receives m_4, sends no messages.
e_5	v_0	–	m_5	–	–	–	v_0 receives and sends no messages.

Our formulation does not require special wakeup messages since processors may be enabled even when none of their incoming edges contain any messages.

2.2. Distributed tasks. A *distributed task* for n processors is defined as a relation T on $X^n \times Y^n$. For example, the task of finding the maximum is the relation $\{((x_0, \ldots, x_{n-1}), (y, \ldots, y)) \mid y = \max_{i=0}^{n-1}\{x_i\}\}$. Let $X_T \subseteq X^n$ be the set of inputs $\vec{x}$ for which there exists an output $\vec{y} \in Y^n$ such that $(\vec{x}, \vec{y}) \in T$.

Let $\mathcal{S}$ be an arbitrary schedule class. A distributed algorithm A is *correct* for input $\vec{x} \in X_T$ and schedule $S \in \mathcal{S}$ if in the execution of A on $\vec{x}$ according to S, all processors terminate and the output $\vec{y}$ satisfies $(\vec{x}, \vec{y}) \in T$. A distributed algorithm A *solves* a distributed task T if A is correct for every input $\vec{x} \in X_T$ and schedule $S \in \mathcal{S}$.

Correctness depends on the task T, the set of private inputs X_T, and the schedule class $\mathcal{S}$. Sometimes, restricting the set X_T drastically changes its complexity. A difficult task might become trivial by severely restricting the inputs. For example, if T is leader election (only one private output is 1 and all the rest are 0), then the task is trivial if X_T is restricted to contain only tuples which have exactly one component with the value 1 and all the rest with 0. The algorithm that writes its private input on its private output without any communication is correct. However, if X_T contains private inputs for which all components are equal, the task becomes impossible [1].

A *cost function* is a mapping from the set of all executions to the natural numbers. Given a distributed algorithm A, an input $\vec{x} \in X_T$, and a schedule $S \in \mathcal{S}$, let $\text{cost}(A, \vec{x}, S)$ denote the cost of the corresponding execution.

We will mainly consider communication costs: *message complexity*—the number of messages sent; and *bit complexity*—the total number of bits sent by all the processors during the execution. However, our discussion is valid for other cost measures as well.

2.3. Average cost of deterministic algorithms. Let T be a distributed task and P be a probability distribution over the input set X_T. The *average cost of a deterministic algorithm A with respect to distribution P and a schedule $S \in \mathcal{S}$* is

$$\text{distribution-cost}(A, S, P) = \boldsymbol{E}_{\vec{x}}(\text{cost}(A, \vec{x}, S), P) = \sum_{\vec{x} \in X_T} P(\vec{x}) \cdot \text{cost } (A, \vec{x}, S).$$

The *average cost of algorithm A with respect to distribution P*, distribution-cost(A, P), is the average cost of the algorithm under the worst possible schedule. However, since the maximum need not exist, we define it to be the supremum over $\mathcal{S}$ of the average cost of the deterministic algorithm A under schedule $S \in \mathcal{S}$.

The *average cost of a task T with respect to distribution P and schedule S*, distribution-cost$_T(S, P)$, is the cost of the best algorithm that solves T under S. Again, since when the set of algorithms is infinite, there may not be a best algorithm, it is defined to be the infimum of the average cost of A with respect to distribution P taken over all uniform distributed algorithms A that solve T.

The *average cost of a task T with respect to distribution P* is the supremum of the average cost of T with respect to distribution P and S taken over all schedules $S \in \mathcal{S}$ and is denoted distribution-cost$_T(P)$. Note that only algorithms that are correct with respect to *every* schedule $S \in \mathcal{S}$ are considered in this definition.

2.4. Randomized algorithms. While the transitions of a deterministic algorithm depend solely on the current state and the messages received, the transitions of a *randomized algorithm* may also depend on the outcome of coin tosses. To simplify notation, we assume that the number of coin tosses performed by each processor of a randomized program in an execution is exactly L. However, the validity of our technique and results does not depend on this assumption.

The Boolean L-tuple ρ_i of results of the L coin tosses of a processor v_i in the execution is called the *private random input* of v_i, and $\vec{\rho} = (\rho_1, \dots, \rho_n) \in \{\{0,1\}^L\}^n$, the n-tuple of private random inputs, is called the *random input* of the execution.

As is customary, we view each processor that runs a randomized algorithm as having access to an additional input tape, called the *random tape.* In addition to the input, each processor is given its own random tape. In each run, the random inputs are chosen uniformly at random so that every one of the 2^{nL} n-tuples of random tapes has the same probability. The "coin-toss" operation of processor v is viewed as accessing the next bit of v's random tape. A random algorithm can thus be "derandomized" by fixing the content of the random tapes. We view this process as follows: for every processor, we replace the random tape by a read-only work tape that contains a fixed binary string (of length L). The operation of reading the next bit of the random tape is replaced by the operation of reading the next bit of this constant tape.

For a randomized algorithm R and $\vec{\rho} \in \{\{0,1\}^L\}^n$, let $R[\vec{\rho}]$ denote the deterministic algorithm resulting from R when for every processor v_i, the operation of reading the next bit from the random tape is replaced by the operation of reading the next bit of $\vec{\rho}_i$.

Let OUTPUT$(R[\vec{\rho}], \vec{x}, S)$ denote the private outputs that result from applying $R[\vec{\rho}]$ under schedule S and input $\vec{x}$. Then R is correct for T if for every $\vec{x} \in X_T$, $\vec{\rho} \in \{\{0,1\}^L\}^n$, and $S \in \mathcal{S}$,

$$(\vec{x}, \text{OUTPUT}(R[\vec{\rho}], \vec{x}, S)) \in T.$$

In §3.3, the definition of correctness is weakened to include correctness with probability 1 and, in §5, to include algorithms that are correct only with probability $\varepsilon < 1$. First, we prove our results for the stronger correctness requirement given above and then generalize it to the weaker definitions.

Since we assume that 0 and 1 are equally likely in each coin toss, each of the 2^{nL} random inputs has equal probability. Therefore, we define the *expected randomized cost of algorithm R for input $\vec{x}$ under schedule S* to be

$$\text{randomized-cost}(R, \vec{x}, S) = \boldsymbol{E}_{\vec{\rho}}(\text{cost}(R[\vec{\rho}], \vec{x}, S)) = 2^{-nL} \sum_{\vec{\rho} \in \{\{0,1\}^L\}^n} \text{cost}(R[\vec{\rho}], \vec{x}, S).$$

The *randomized cost of algorithm R under schedule S* is

$$\text{randomized-cost}(R, S) = \sup_{\vec{x} \in X_T} \text{randomized-cost}(R, \vec{x}, S),$$

and *the randomized cost of algorithm R* is

$$\text{randomized-cost}(R) = \sup_{S \in \mathcal{S}} \text{randomized-cost}(R, S).$$

Finally, let randomized-cost$_T$, *the randomized cost of the task T under schedule class $\mathcal{S}$*, be the infimum of randomized-cost(R) over all randomized algorithms R that solve T. Note that in the definitions of this subsection, the expectations are taken over the coin tosses with respect to the worst possible input $\vec{x} \in X_T$.

2.5. Relationship to other models. Since we are interested in lower bounds, we have allowed the computational capabilities of the processors to be as strong as possible and restricted the schedule class. The schedule classes we allowed are limited in that they allow only FIFO discipline on the edges and the reception of only one message at a time. Since a lower bound exhibits the existence of a schedule on which the algorithm behaves badly, this schedule belongs to any schedule class that contains ours; therefore, the lower bound holds also for the more general classes.

However, lower bounds would not necessarily hold for more restricted schedule classes. To prove a lower bound on randomized algorithms under such a schedule class, one should explicitly restrict the discussion to the *same* schedule class.

Some examples of schedule classes are the following:

1. *synchronous schedules*: the processors are enabled in lock step;
2. *fair schedules*: the schedules in which in a every infinite execution each edge occurs infinitely often.

The situation is reversed when considering the computational power of the processors. A lower bound that holds for processors with strong computational power also holds for more restricted processors. Additional models can be simulated by restricting the class of allowable transition tables. For example, to model a message-driven setup, it suffices that the state remains unchanged unless the in-buffer is nonempty. In order to simulate wakeup messages, we allow a state transition from the initial state even when the buffers are empty.

3. Yao's lemma.

3.1. Restating Yao's lemma. In this subsection, we restate Yao's lemma to fit our needs. For this, we need the following definition:

We may view a (uniform) randomized algorithm R as a mapping of each pair $(\vec{x}, S)$ of input and schedule to a probability distribution over the executions of R on input $\vec{x}$ under schedule S. A *canonical representation of* R is a probability distribution over a set of deterministic algorithms $\mathcal{A}$ such that

(a) each algorithm $A \in \mathcal{A}$ is uniform;

(b) for each input $\vec{x}$, each schedule S, and each execution ϵ, the probability that R on input $\vec{x}$ performs execution ϵ is equal to the probability that on the same input $\vec{x}$ and schedule S, an algorithm A chosen at random from $\mathcal{A}$ performs execution ϵ.

We restate Yao's lemma to include an appropriate consideration of the schedule and an explicit stating of the assumption that a model must fulfill in order to make the lemma valid.

LEMMA 3.1 (Yao). *Let $\mathcal{S}$ be a schedule class, $S \in \mathcal{S}$ a schedule, T a distributed task over the inputs $X_T \subseteq X^n$, R a randomized algorithm that solves T, and $\mathcal{A}$ a canonical representation of R. Then for every probability distribution P over X_T, there is a deterministic algorithm $A \in \mathcal{A}$ such that*

$$\text{distribution-cost}(A, S, P) \leq \text{randomized-cost}(R, S).$$

3.2. Main results. The counterexample in §1 demonstrated that, in general, Yao's inequality need not hold. However, we now show that if we restrict ourselves to componentwise-distinct inputs, Yao's lemma can be extended to the distributed case.

First, let us examine the representation of random algorithms that is traditionally used (for example in the PRAM model) for implementing Yao's inequality, sometimes implicitly. Recall that for $\vec{\rho} \in \{\{0,1\}^L\}^n$, $R[\vec{\rho}]$ is the deterministic algorithm which results from R if in every processor v_i, every coin-toss operation is replaced by reading the next bit of ρ_i. The traditional technique represents a random algorithm R by the set

$$\mathcal{R} = \left\{R[\vec{\rho}] \mid \vec{\rho} \in \{\{0,1\}^L\}^n\right\}$$

under the uniform distribution on $\{0,1\}^{Ln}$.

The problem of using $\mathcal{R}$ in our distributed model is that it does *not* consist only of uniform algorithms since for nearly all $\vec{\rho} = (\rho_1, \ldots, \rho_n) \in \{\{0,1\}^L\}^n$, $i \neq j$ implies that $\rho_i \neq \rho_j$ and therefore $R[\rho_i]$, the transition table of v_i, is different from that of v_j. Thus we cannot apply Yao's lemma using this representation since the first requirement of a canonical representation—that of uniformity—is violated.

THEOREM 3.2. *Let $\mathcal{S}$ be a schedule class, $S \in \mathcal{S}$, T a distributed task over an input set $X_T \subseteq X^n$ consisting of only componentwise-distinct inputs, P a probability distribution over X_T, and R a randomized distributed algorithm that solves T. Then there exists a deterministic algorithm D that solves T such that*

$$\text{distribution-cost}(D, S, P) \leq \text{randomized-cost}(R, S) \leq \text{randomized-cost}(R).$$

Proof. First, we show that for every randomized algorithm R, there exists a canonical representation $\mathcal{A}$.

Let f be a bijection from $X \times \mathbb{N}$ to $\mathbb{N}$. (For example, if $X = \mathbb{N}$, $f(x,i) = \frac{1}{2}(x+i)(x+i+1)+i$.)

Let $\{0,1\}^\omega$ denote the set of all infinite sequences over $\{0,1\}$). For each $\sigma \in \{0,1\}^\omega$, let $R_f[\sigma]$ be the deterministic algorithm in which the state diagram of each processor is identical to R's, except for the following changes:

1. the string σ is a constant of the program of $R_f[\sigma]$;

2. every read operation from the random tape is replaced by a read operation of σ, i.e., when in R a processor reads the ith bit from its random tape, in $R_f[\sigma]$, the processor reads bit $f(x,i)$ of σ, where x is the private input of the processor.

Let

$$\mathcal{A} = \{R_f[\sigma] | \sigma \in \{0,1\}^\omega\}.$$

We now show that $\mathcal{A}$ with the uniform distribution on $\{0,1\}^\omega$ is a canonical representation of R.

To show (a), for each σ, $R_f[\sigma]$ is a uniform algorithm since all the processors have the same state diagram. In the random algorithm, two processors may have acted differently on the same inputs because their random tapes were different. However, in $R_f[\sigma]$, there is no random tape—it was replaced by σ. The sequence of bits of σ considered by each processor is a function of the processor's private input—not its index. However, the private input is not part of the program. If, for example, processor i were given the private input of processor j, processor i would consider the same bits previously considered by processor j, and thus its actions would be exactly identical to those of processor j, i.e., both processors have the same state diagram, i.e., the algorithm is uniform.

To show (b), fix the input $\vec{x} = (x_1, \ldots, x_n) \in X_T$ and a schedule S. The execution now depends only on the random inputs. We say that a coin toss $\vec{\rho} \in \{\{0,1\}^L\}^n$ *implies* execution ϵ if ϵ occurs when R is run with random input $\vec{\rho}$. Since each of the 2^{nL} random sequences is equally likely, the probability of execution ϵ is equal to the number of tosses that imply ϵ divided by 2^{nL}. Define an equivalence relation $\cong_{\vec{x}}$ on $\{0,1\}^\omega$ such that $\sigma \cong_{\vec{x}} \sigma'$ if for $i = 1, \ldots, L$ and $j = 1, \ldots, n$,

$$\sigma_{f(x_j,i)} = \sigma'_{f(x_j,i)}.$$

Under the uniform distribution on $\{0,1\}^\omega$, each of the equivalence classes has probability 2^{-nL}.

For each input $\vec{x}$ as above, we define a 1–1 correspondence between the equivalence classes above and random inputs $\vec{\rho} \in \{\{0,1\}^L\}^n$: a random input $\vec{\rho} = (\rho_1, \ldots, \rho_n)$ corresponds to σ if for $i = 1, \ldots, L$ and $j = 1, \ldots, n$, $\rho_{j,i} = f(x_j, i)$. If σ belongs to an equivalence class that corresponds to $\vec{\rho}$, then the execution of $R_f[\sigma]$ is equal to that of R with random inputs $\vec{\rho}$. Given an equivalence class $C \subseteq \{0,1\}^\omega$, the probability of choosing $\sigma \in C$ is equal to 2^{-nL}, and that is equal to the probability of choosing any random input. In particular, it is equal to choosing the random input which corresponds to C. Consequently, the probability of choosing an algorithm $R_f[\sigma] \in \mathcal{A}$ whose execution is ϵ equals the probability that the execution of the randomized algorithm R is ϵ.

We still need to show that each algorithm $R_f[\sigma] \in \mathcal{A}$ solves T. Since R is correct for T, for every input $\vec{x}$ and every schedule $S \in \mathcal{S}$, every execution of R on $\vec{x}$ under S produces a correct result. Since every execution of $R_f[\sigma]$ corresponds to some execution of R, we must have that every execution of $R_f[\sigma]$ must produce a correct result; hence $R_f[\sigma]$ solves T.

We may now apply Lemma 3.1 to prove the theorem. □

As a corollary of Theorem 3.2, we have the following.

COROLLARY 3.3. *Let T be a distributed task over an input set $X_T \subseteq X^n$ consisting of only componentwise-distinct inputs, $\mathcal{S}$ a schedule class, and P a probability distribution over X_T. Then for every $S \in \mathcal{S}$,*

$$\text{distribution-cost}_T(S,P) \leq \text{randomized-cost}_T(S,P) \leq \text{randomized-cost}_T.$$

Note that Corollary 3.3 can be used to obtain lower bounds on the randomized complexity of a distributed task even if its input set does not consist solely of componentwise-distinct inputs because a lower bound for a restricted set of inputs implies a lower bound for a superset.

3.3. Randomized algorithms that are correct with probability 1. We can generalize Theorem 3.2 and Corollary 3.3 to also hold for randomized algorithms that are correct with probability 1 (i.e., for every (x, S), there is probability 0 that the randomized algorithm errs). This can occur only if the number of coin tosses is unbounded. Hence we abandon our methodological assumption that this number is finite.

An additional effort is needed only to show that if $\mathcal{A}$ is our canonical representation of a randomized algorithm R that is correct with probability 1, then there exists a canonical representation $\mathcal{A}'$ for R such that all $A \in \mathcal{A}'$ solve T. Since the number of possible schedules might be uncountable, this last result is not immediate. However, this result can be proved for all of the cost functions that we considered. We sketch below the proof for the case where the cost function is the number of messages sent.

This result will follow from the next two lemmas. Lemma 3.4 implies that for every randomized algorithm R that is correct with probability 1, there exists a randomized algorithm R' that is also correct with probability 1, has the same complexity, and, for some finite $M > 0$, never sends more than M messages.

This implies that the complexity of a task cannot be affected by considering algorithms that do not have a finite bound on the number of messages they send. Thus, without loss of generality, such algorithms may be ignored.

LEMMA 3.4. *Let R be a randomized algorithm that solves T with probability* 1. *Then for every $\delta > 0$, there is a randomized algorithm R^δ that solves T with probability* 1 *such that*

(a) *every execution of R^δ terminates after at most $n^4(1+\delta^{-1})$ messages are sent, and*

(b) distribution-cost$(R^\delta) \leq (1 + \delta)$randomized-cost$(R)$.

Proof (outline). For $\vec{x} \in X_T$ and $I \subseteq \{1, \ldots, n\}$, $\vec{y}^I = (y_1^I, \ldots, y_n^I)$ is a *partial output* if there exists $\vec{y} \in Y^n$ for which $(\vec{x}, \vec{y}) \in T$, $\vec{y}_i^I = y_i$ for $i \in I$ and $y_i^I = \perp \notin Y$ otherwise. Let g be a function that maps every input $\vec{x}$ and partial output $\vec{y}^I$ to a full output $\vec{y}\,'$ such that $(\vec{x}, \vec{y}\,') \in T$ and $\vec{y}^I$ and $\vec{y}\,'$ agree on I. (If y^I is not a partial output of $\vec{x}$, i.e., there exists no such $\vec{y}\,'$, then $g(\vec{x}, \vec{y}^I)$ is arbitrary.)

Given a randomized algorithm R and $\delta > 0$, R^δ is defined as follows. Every processor v_i simulates R until v_i sends n^3/δ messages and then, if v_i did not terminate the algorithm, it stops executing R and broadcasts its private input. Also, upon first receiving a broadcast message, a processor stops its regular execution and broadcasts its private input (and its private output if it had already been computed). Let I consist of the processors which computed their private output before participating in the broadcast. Upon receiving the broadcasts from all other processors, processor v_i $(i \notin I)$ computes $\vec{y}\,' = g(\vec{x}, \vec{y}^I)$ and outputs y_i'.

To implement the broadcast, each time a processor gets new information, it sends it to all its adjacent vertices. Thus each edge is traversed at most $2(n-1)$ times, and the message complexity is at most $2(n-1)|E| < n^3$. (If the network contains parallel edges between two vertices v_i and v_j, then each message from v_i to v_j is sent on only one of these parallel edges.)

Since R is correct with probability 1, with probability 1, y^I is a partial solution, and R^δ extends it to a full solution $\vec{y}\,'$. Thus R^δ also solves T with probability 1.

The lemma follows since in every execution of R^δ, every processor sends at most n^3/δ messages before switching to algorithm A^I. If, during an execution of R^δ, a processor switched to algorithm A^I, then the number of messages sent by R during that execution, m, was at least n^3/δ. (b) follows since the number of messages sent by algorithm A^I is at most $n^3 = \delta(n^3/\delta) \leq \delta m$. □

LEMMA 3.5. *Let $M > 0$. Let R be a randomized algorithm that solves T with probability* 1 *and sends no more than M messages, and let $\mathcal{A}$ be a canonical representation of R. Then with probability* 1, *an algorithm $A \in \mathcal{A}$ solves T. Also, there exists a canonical representation $\mathcal{A}'$ of R such that every $A \in \mathcal{A}'$ solves T.*

Proof (sketch). The bound M on the number of messages allows us to assume that the schedule class $\mathcal{S}$ is countable.

For input $\vec{x}$ and schedule S, let $\mathcal{ERR}_{\vec{x},S}$ be the set of algorithms which err on $\vec{x}$ under schedule S. Since for each $\vec{x}$ and S, the probability that an algorithm chosen at random from $\mathcal{A}$ errs is 0, for each $(\vec{x}, S)$, the probability of $\mathcal{ERR}_{\vec{x},S}$ is 0. Let $\mathcal{ERR} = \bigcup_{\vec{x},S} \mathcal{ERR}_{\vec{x},S}$. Since both X_T and $\mathcal{S}$ are countable, the probability of choosing an algorithm $A \in \mathcal{ERR}$ is $P(\mathcal{ERR}) \leq \sum_{\vec{x},S} P(\mathcal{ERR}_{\vec{x},S}) = 0$ (a countable sum of zeroes). Thus the probability that an algorithm chosen at random from $\mathcal{A}$ errs on some $\vec{x}$ for some schedule S is 0.

The canonical representation $\mathcal{A}'$ is obtained from $\mathcal{A}$ by removing all the algorithms of $\mathcal{ERR}$. □

4. Applications. Like Yao's original method, our results suggest the following technique for proving lower bounds on the randomized complexity of distributed tasks with componentwise-distinct inputs:

1. Find a probability distribution P over the set of componentwise-distinct inputs, and find a schedule S for which a lower bound can be shown on distribution-cost(T, S, P), the average (with respect to distribution P) cost of deterministic distributed algorithms that solve T under schedule S. (Note that this lower bound has to hold only for deterministic algorithms that are correct for every $S \in \mathcal{S}$. This property is important for proving deterministic lower bounds.)

2. Apply Corollary 3.3 to conclude that this lower bound holds also for the randomized complexity of the same task.

Our technique can sometimes be used even if we only have a lower bound on the worst case. When there is a single componentwise-distinct input $\vec{x} \in X_T$ for which every deterministic algorithm satisfies the lower bound, choose a distribution P that gives $\vec{x}$ probability (close to) 1 (and probability (almost) 0 to $X_T - \{\vec{x}\}$). (This technique is used in Theorem 4.2 below.)

In 1988, Bodlaender [3] proved an $\Omega(n \log n)$ lower bound on the average message complexity for finding the maximum id in an asynchronous ring of processors that holds even if the ring is bidirectional and even if the ring size n is known to the processors in advance, provided that the set of possible ids is at least $2n^3$. The same lower bound with different parameters was also published by P. Duris and Z. Galil [4] in 1987.

Bodlaender's proof satisfies both the requirements of 1:

(a) In Bodlaender's task, the input is an n-tuple of id's, i.e., the private inputs are distinct.

(b) Bodlaender stated his lower bound for the class of asynchronous schedules. However, in the proof, he showed a specific schedule on which this lower bound holds.

Thus we may apply Corollary 3.3 to Bodlaender's result to show the following lower bound.

THEOREM 4.1. *Let T be the task of finding the maximum id in a bidirectional asynchronous ring of n processors, where there are at least $2n^3$ possible ids. Then the* *randomized message complexity of T is at least $\Omega(n \log n)$. This lower bound holds even if the ring size n is known to the processors in advance.*

In 1985, Frederickson and Lynch [6] showed that the problem of finding the maximum id in a synchronous bidirectional ring of n processors has an $\Omega(n \log n)$ lower bound on the worst-case message complexity when the algorithms are assumed to use comparison only. Their proof constructs a permutation π of $\{1, \dots, n\}$ such that if the id of processor i is π_i, then every correct algorithm (which uses comparisons only) requires $\Omega(n \log n)$ messages. As in the counterexample to Yao's lemma, we construct a distribution P that gives probability 1 to the input $\vec{x} = \pi$ and probability 0 to all other inputs. Frederickson and Lynch's proof shows that distribution-cost$(T, S, P) = \Omega(n \log n)$ messages. Since the inputs are a permutation, they are componentwise distinct. Hence we get the following theorem.

THEOREM 4.2. *The problem of finding the maximum id in a synchronous bidirectional ring of n processors has an $\Omega(n \log n)$ lower bound on the randomized message complexity when the algorithms are assumed to use comparison only.*

Note that the last two lower bounds hold even if the randomized algorithms are allowed to err with probability 0.

5. Bounded error.

In his paper, Yao also presented an inequality for the probabilistic complexities when a bounded error is allowed. With our technique, this inequality can also be extended to the distributed model.

Let A be a deterministic distributed algorithm that solves task T, $\vec{x} \in X_T$, and $S \in \mathcal{S}$ be a schedule. Define

$$\mathrm{ERR}(A, T, \vec{x}, S) = \begin{cases} 1 & \mathrm{OUTPUT}(A, \vec{x}, S)) \notin T, \\ 0 & \text{otherwise}, \end{cases}$$

where $\mathrm{OUTPUT}(A, \vec{x}, S)$ is the output of A on input $\vec{x}$ under schedule S.

Let P be a probability distribution over X_T and let $\delta \geq 0$. We overcome measurability problems by using the assumption that X_T is countable. *A solves a task T with error δ under schedule S* if the expected error satisfies

$$\boldsymbol{E}_{\vec{x}}(\mathrm{ERR}(A, T, \vec{x}, S), P) = \sum_{\vec{x} \in X_T} P(\vec{x}) \cdot \mathrm{ERR}(A, T, \vec{x}, S) \leq \delta.$$

Let distribution-cost$^{\delta}_{T}(S, P)$, the *average cost of task T with error δ with respect to distribution P and schedule S*, be the infimum of distribution-cost(A, S, P) taken over all the deterministic algorithms A that solve T for schedule S with error δ.

Consider a randomized algorithm R. If we fix the input $\vec{x}$, then $\mathrm{ERR}(R[\vec{\rho}], T, \vec{x}, S)$ is a function of $\vec{\rho} \in (\{0,1\}^{\omega})^n$. Moreover, we show the following.

LEMMA 5.1. *For every $\vec{x} \in X_T$, $\mathrm{ERR}(R[\vec{\rho}], T, \vec{x}, S)$ is a measurable function of $\vec{\rho}$ over $(\{0,1\}^{\omega})^n$.*

Proof. Consider a vector of finite sequences $\tau \in (\{0,1\}^k)^n$. Let $\mathrm{CONT}(\tau)$ consist of all the infinite continuations of τ, i.e.,

$$\mathrm{CONT}(\tau) = \{\sigma \in (\{0,1\}^{\omega})^n \ : \ \tau_i[j] = \sigma_i[j], i = 1, \dots, n, \ j = 1, \dots, k\}.$$

Obviously, for every such τ, $\mathrm{CONT}(\tau)$ is measurable.

For $\vec{x} \in X_T$, $\vec{\rho} \in (\{0,1\}^{\omega})^n$, let $\ell(R, \vec{\rho}, \vec{x}, S) \leq \infty$ denote the largest number of random bits accessed by any processor running R with random tapes $\vec{\rho}$ and input $\vec{x}$ under schedule S. Let

$$\mathrm{HALT}_k = \{\vec{\rho} \in (\{0,1\}^{\omega})^n \ : \ \ell(R, \vec{\rho}, \vec{x}, S) = k\}.$$

If $\vec{\rho}$ and $\vec{\rho}' \in \text{CONT}(\tau)$ and $\vec{\rho} \in \text{HALT}_k$, then

1. $\vec{\rho}' \in \text{HALT}_k$, and
2. $\text{ERR}(R[\vec{\rho}], T, \vec{x}, S) = \text{ERR}(R[\vec{\rho}'], T, \vec{x}, S)$.

Thus there exists a finite set of sequences $I_k = \{\tau^1, \dots, \tau^q\}$ such that $\text{HALT}_k = \bigcup_{\tau^j \in I_k} \text{CONT}(\tau^j)$. Since HALT_k is a finite union of measurable sets, HALT_k is measurable.

Let $\text{CORRECT}_k \subseteq \text{HALT}_k$ be the set of all sequences $\vec{\rho} \in \text{HALT}_k$ for which $R[\vec{\rho}]$ is correct for $\vec{x}$ (i.e., $\text{ERR}(R[\vec{\rho}], T, \vec{x}, S) = 0$). $\text{ERR}(R[\vec{\rho}], T, \vec{x}, S)$ is constant on every $\tau^i \in I_k$. Let $I_k^0 \subseteq I_k$ consist of the sequences of I_k for which R is correct. $\text{CORRECT}_k = \bigcup_{\tau^j} \in I_k^0$ and is measurable since it is a finite union of a measurable sets. $\text{CORRECT} = \bigcup_{k \geq 0} \text{CORRECT}_k$ is also measurable.

$\text{ERR}(R[\vec{\rho}], T, \vec{x}, S)$ is a measurable function since $1 - \text{ERR}(R[\vec{\rho}], T, \vec{x}, S)$ is the characteristic function of the measurable set CORRECT. □

Since $\text{ERR}(R[\vec{\rho}], T, \vec{x}, S)$ is measurable, we may define its expectation $\boldsymbol{E}_{\vec{\rho}}(\text{ERR}(R[\vec{\rho}], T, \vec{x}, S))$ over the coin tosses ($\vec{\rho} \in (\{0,1\}^\omega)^n$). A randomized distributed algorithm *solves a task T with error δ under schedule S* if for every input $\vec{x}$,

$$\boldsymbol{E}_{\vec{\rho}}(\text{ERR}(R[\vec{\rho}], T, \vec{x}, S)) \leq \delta.$$

A randomized algorithm R *solves T with error δ* if for every $S \in \mathcal{S}$, R solves T with error δ under S. $\text{randomized-cost}_T^\delta(S)$, *the randomized cost of task T with error δ*, is the infimum of $\text{randomized-cost}_T(R[\vec{\rho}], S)$ taken over all the randomized algorithms that solve T under schedule S with error δ.

Using Yao's result concerning Monte Carlo algorithms and the same technique that was used to prove Theorem 3.2, we obtain the following.

THEOREM 5.2. *Let T be a distributed task over a countable input set $X_T \subseteq X^n$ consisting of only componentwise distinct inputs, S a schedule, and P a probability distribution over X_T. Then for every $0 \leq \delta \leq \frac{1}{2}$,*

$$\frac{1}{2}\text{distribution-cost}_T^{2\delta}(S, P) \leq \text{randomized-cost}_T^\delta(S).$$

Proof. Let R be a randomized algorithm that solves the task T within error δ under schedule S. For every infinite binary sequence $\sigma \in \{0,1\}^\omega$, let $R_f[\sigma]$ be the deterministic algorithm which results if processor j uses the $f(x_j, i)$th bit of σ as the outcome of the ith coin toss. As before, $\{R_f[\sigma] \mid \sigma \in \{0,1\}^\omega\}$ is a canonical representation and $\boldsymbol{E}_\sigma(\text{cost}(R_f[\sigma], \vec{x}, S)) = \boldsymbol{E}_{\vec{\rho}}(\text{cost}(R[\vec{\rho}], \vec{x}, S))$.

Since R solves T within error δ, for every $\vec{x} \in X_T$,

$$\boldsymbol{E}_{\vec{\rho} \in (\{0,1\}^\omega)^n}(\text{ERR}(R[\vec{\rho}], T, \vec{x}, S)) \leq \delta.$$

Given a distribution P on the inputs, the error probability of $R_f[\vec{\rho}]$, $\boldsymbol{E}_{\vec{x}}(\text{ERR}(R[\vec{\rho}], T, \vec{x}, S), P) = \sum_{\vec{x} \in X_T} P(\vec{x}) \cdot \text{ERR}(R[\vec{\rho}], T, \vec{x}, S)$, is measurable since it is an infinite sum of measurable functions [8, Thm. 1.27, p. 22]. Its expectation over $(\{0,1\}^\omega)^n$ satisfies

$$\begin{aligned} \boldsymbol{E}_{\vec{\rho} \in (\{0,1\}^\omega)^n}(\boldsymbol{E}_{\vec{x}}(\text{ERR}(R[\vec{\rho}], T, \vec{x}, S), P)) &= \boldsymbol{E}_{\vec{x}}(\boldsymbol{E}_{\vec{\rho} \in (\{0,1\}^\omega)^n}(\text{ERR}(R[\vec{\rho}], T, \vec{x}, S), P)) \\ &= \boldsymbol{E}_{\vec{x}}(\boldsymbol{E}_{\sigma \in (\{0,1\}^\omega)}(\text{ERR}(R_f[\sigma], T, \vec{x}, S), P)) \\ &\leq \boldsymbol{E}_{\vec{x}}(\delta) = \delta. \qquad (1) \end{aligned}$$

(The measurability of $\text{ERR}(R_f[\sigma], T, \vec{x}, S)$ over $\sigma \in \{0,1\}^\omega$ is similar to Lemma 5.1.)

Let $C \subseteq \{0,1\}^\omega$ denote the set of sequences for which $\boldsymbol{E}_{\vec{x}}(\mathrm{ERR}(R_f[\sigma], T, \vec{x}, S)) \leq 2\delta$. Since $\mathrm{ERR}(R_f[\sigma], T, \vec{x}, S)$ is measurable, so is $\boldsymbol{E}_{\vec{x}}(\mathrm{ERR}(R_f[\sigma], T, \vec{x}, S)) = \sum_{\vec{x} \in X_T} P(\vec{x}) \cdot \mathrm{ERR}(R_f[\sigma], T, \vec{x}, S)$ (a sum of measurable functions). Hence by [8, Exercise 5, p. 32], C is measurable.

Equation (1) implies that $P(C) \geq \frac{1}{2}$. Hence there exists a sequence $\sigma^* \in C$ such that

$$\text{distribution-cost}(R_f[\sigma^*], S, P) \leq 2\text{randomized-cost}(R, S).$$

Since $\sigma^* \in C$, $\boldsymbol{E}_{\vec{x}}(\mathrm{ERR}(R_f[\sigma^*], T, \vec{x}, S), P) \leq 2\delta$. Thus if R has expected cost randomized-cost$_T^{\delta}(S)$, we have exhibited a deterministic algorithm $R_f[\sigma^*]$ which errs with probability at most 2δ and its expected cost is at most 2randomized-cost(R, S)). □

Another result by Yao connects the randomized cost with small error to the average cost with no error. Combining our techniques with those of Yao, we can extend these ideas to the distributed model.

THEOREM 5.3. *Let T be a distributed task over a finite input set $X_T \subset X^n$ consisting only of componentwise distinct inputs, S a schedule, and P a probability distribution over X_T. Then for every $0 \leq \delta \leq 1$,*

$$\begin{aligned}(1-\delta)\text{distribution-cost}_T^0(S, P) &\leq \text{randomized-cost}_T^{(\delta/|X_T|)}(S)\\ &\leq \text{randomized-cost}_T^{(\delta/|X_T|)}.\end{aligned}$$

As an application, we can extend a result by Frederickson and Lynch [6] concerning deterministic worst-case complexity of synchronous algorithms.

COROLLARY 5.4. *Let T be the task of electing a leader in a synchronous ring of size n, t be a positive integer, and $0 < \delta < 1$. If the set of inputs X_T is a sufficiently large finite set, then the expected number of messages required by any randomized algorithm that solves T within t rounds with bounded error $\delta/|X_T|$ requires $\Omega(n \log n)$ messages.*

6. An open problem. We have shown that in the distributed model for every schedule S,

$$\text{distribution-cost}_T(S, P) \leq \text{randomized-cost}_T(S) \leq \text{randomized-cost}_T,$$

provided the inputs are componentwise distinct.

Note that we can only state that for every schedule S, there exists a deterministic algorithm $A(S)$ such that distribution-cost$(A(S), S, P) \leq$ randomized-cost$_T(S)$. It remains an open question whether there exists a single deterministic algorithm for which for all schedules S the inequality holds. In other words, while we have shown that

$$\text{distribution-cost}_T(P) = \sup_S \inf_A \text{distribution-cost}(A, S, P) \leq \text{randomized-cost}_T,$$

it remains open whether

$$\inf_A \sup_S \text{distribution-cost}(A, S, P) \leq \text{randomized-cost}_T.$$

For the special case where the system can be modeled by a single schedule (i.e., the set $\mathcal{S}$ of schedules is a singleton), Corollary 3.3 indeed implies the last inequality. This happens, for example, when modeling a synchronous system.

However, as we have seen in §4, our results are sufficient to show nontrivial optimal lower bounds for randomized complexity even for the general asynchronous case.

Acknowledgments. It is a pleasure to thank A. Herzberg for his helpful advice and S. Ben-David and A. Fiat for helpful discussions.

REFERENCES

[1] D. ANGLUIN, *Local and global properties in networks of processes*, in Proc. 12th ACM Symposium on the Theory of Computing (STOC), Association for Computing Machinery, New York, 1980, pp. 82–93.

[2] H. ATTIA, M. SNIR, AND M. WARMUTH, *Computing on the anonymous ring*, in Proc. 4th Annual ACM Symposium on Principles of Distributed Computing (PODC), Association for Computing Machinery, New York, 1985, pp. 196–203.

[3] H. L. BODLAENDER, *New lower bound techniques for distributed leader finding and other problems on rings of processors*, Theoret. Comput. Sci., 81 (1991), pp. 237–256.

[4] P. DURIS AND Z. GALIL, *Two lower bounds in asynchronous distribute computation*, in Proc. 28th IEEE Symposium on the Foundations of Computer Science (FOCS), IEEE Computer Society Press, Los Alamitos, CA, 1987, pp. 326–330.

[5] F. E. FICH, F. MEYER AUF DER HEIDE, P. RAGDE, AND A. WIGDERSON, *One, two, three...infinity: Lower bounds for parallel computation*, in Proc. 17th ACM Symposium on the Theory of Computing (STOC), Association for Computing Machinery, New York, 1985, pp. 48–58.

[6] G. N. FREDERICKSON AND N. A. LYNCH, *Electing a leader in a synchronous ring*, J. Assoc. Comput. Mach., 34 (1987), pp. 98–115.

[7] A. ITAI AND M. RODEH, *Probabilistic methods for breaking symmetry in distributed networks*, Inform. and Comput., 88 (1990), pp. 60–87.

[8] W. RUDIN, *Real and Complex Analysis*, 3rd ed., McGraw–Hill, New York, 1966.

[9] A. C. YAO, *Probabilistic computations: Towards a unified measure of complexity*, in Proc. 18th IEEE Symposium on the Foundations of Computer Science (FOCS), IEEE Computer Society Press, Los Alamitos, CA, 1977, pp. 222–227.

SIAM J. COMPUT.
Vol. 25, No. 6, pp. 1268–1280, December 1996

007

LEARNING BEHAVIORS OF AUTOMATA FROM MULTIPLICITY AND EQUIVALENCE QUERIES*

FRANCESCO BERGADANO† AND STEFANO VARRICCHIO‡

Abstract. We consider the problem of identifying the behavior of an unknown automaton with multiplicity in the field Q of rational numbers (Q-automaton) from multiplicity and equivalence queries. We provide an algorithm which is polynomial in the size of the Q-automaton and in the maximum length of the given counterexamples. As a consequence, we have that Q-automata are probably approximately correctly learnable (PAC-learnable) in polynomial time when multiplicity queries are allowed. A corollary of this result is that regular languages are polynomially predictable using membership queries with respect to the representation of unambiguous nondeterministic automata. This is important since there are unambiguous automata such that the equivalent deterministic automaton has an exponentially larger number of states.

Key words. PAC-learning, exact ientification, learning from examples, learning from queries, equivalence queries, multiplicity queries, membership queries, multiplicity automata, probabilistic automata, unambiguous nondeterministic automata

AMS subject classifications. 68Q68, 68Q70, 68Q75, 68T05

1. Introduction.

Learning automata from examples and from queries has been extensively investigated in the past, and important results have been obtained recently. Early on, it was noticed that the problem of exactly identifying a minimum automaton consistent with given data is NP-complete [10]. Similar results may be proved for regular expressions [1]. Even simply approximating the minimum consistent deterministic finite automaton (DFA) problem is not feasible [14]. Gold [10] proves that polynomial identification in the limit is still possible in the sense that an inductive inference machine will take polynomial time when processing a new example. However, this may seem unsatisfactory since the number of examples needed may be arbitrarily large. A natural direction generally followed in machine learning (see Chapter 2.3 in [6] and references therein) was to consider a learner who did not just passively receive data but who was able to ask queries.

Some questions—called *membership queries*—may consist of asking an oracle whether a particular string belongs to the target language. Angluin [2] proved that if we start from a set of strings that lead to every reachable state in the target automaton, a polynomial number of membership queries is sufficient for exact identification. However, if such a set of strings is not available, even if we know the size n of the target automaton, the number of queries needed is exponential in n.

Another possibility is found in *equivalence queries*: asking an oracle whether a guess is correct and obtaining a counterexample if it is not. We shall also assume that the counterexamples have a maximum length m. It may be shown [5] that there is no polynomial-time algorithm to exactly identify automata from equivalence queries only. However, there is a polynomial algorithm if both equivalence and membership queries are used [3].

* Received by the editors November 1, 1993; accepted for publication (in revised form) February 14, 1995.

† Dimartimento di Informatica, Università di Torino, 185 Corso Svizzera, 10149 Torino, Italy (bergadan@di.unito.it). The research of this author was supported by the European Union under IV Framework ESPRIT, Long Term Research Project "ILP2" (contract 20237).

‡ Dipartimento di Matematica Pura ed Applicata, Università dell'Aquila, Via Vetoio, 67010 L'Aquila, Italy (varricch@univaq.it). The research of this author was partly supported by the Italian Minister of Universities and by ESPRIT–EBRA project ASMICS (contract 6317).

Equivalence and membership queries then seem to be a necessary requirement for learning deterministic finite-state automata. It remains to be seen if stronger formalisms may be learned under the same framework. Following preliminary results reported in [7], this paper gives a positive answer in the direction of behaviors of nondeterministic finite-state automata, i.e., functions that assign to every string the number of its accepting paths in a nondeterministic finite-state acceptor. Such functions, as defined in the next section, will be described in the more general framework of automata with multiplicity.

We introduce the notion of a *multiplicity query*. In the case of a nondeterministic automaton, a multiplicity query returns the number of accepting paths for a given string. We show that behaviors of automata with multiplicity may be identified in polynomial time with multiplicity and equivalence queries. This implies that they are probably approximately correctly learnable (PAC-learnable) with multiplicity queries. If we restrict the result to unambiguous nondeterministic automata, multiplicity queries must return either 0 or 1 and reduce to membership queries. As a consequence of our main result, we may then PAC-learn with membership queries a representation of a regular language L in polynomial time with respect to the size of an unambiguous nondeterministic automaton that accepts L. This is an improvement over the result of Angluin [3] because there are unambiguous automata such that the equivalent DFA has an exponentially larger number of states [16]. However, it must be noted that the learned representation of the regular language is not an unambiguous nondeterministic automaton. Therefore, unambiguous nondeterministic finite automata (NFAs) are only shown to be PAC-predictable with membership queries. Our main result also applies to probabilistic automata. In that case, the multiplicity of a given string represents the probability of accepting that string. Again, our main result implies the PAC-predictability of probabilistic automata with multiplicity queries of this kind. This is an improvement over the results of Tzeng [17], where stronger queries are needed, giving the probability of reaching a given state with a given string.

2. Multiplicity automata.

Automata with multiplicity, also called multiplicity automata, are the most important generalizations of classical automata theory. In recent years, their significant development has helped in solving old problems in automata theory. In [11], using multiplicity automata, the decidability of the equivalence problem for deterministic multitape automata has been solved; in [18], a similar result has been shown for unambiguous regular languages in a free partially commutative monoid.

Let M be an NFA. We can consider the so-called *behavior* of M, which is the map that associates with any word the number of its different accepting paths. More generally, we can assign a multiplicity to the initial states, the final states, and the edges of the automaton so that the corresponding behavior must take into account the assigned multiplicities. In this way, we can construct a theory which is general enough to contain classical and probabilistic automata as particular objects. Multiplicity automata have been extensively studied in theoretical computer science, and we refer to [8], [9], and [15]; here we recall some notation and definitions.

Let K be a field and A^* be the free monoid over a finite alphabet A; we consider the set K^{A^*} of all the applications $S : A^* \to K$. An element $S \in K^{A^*}$ will be called a *K-subset* of A^* or simply a *K-set*. Following the standard notation on K-sets, for any $S \in K^{A^*}$ and $u \in A^*$, we will denote $S(u)$ by (S, u). In what follows, we shall consider $\mathbb{Q}$-sets, where $\mathbb{Q}$ denotes the field of rational numbers.

We denote by $\mathbb{Q}^{n\times n}$ the set of all square $n \times n$ matrices with entries in $\mathbb{Q}$. We shall consider $\mathbb{Q}^{n\times n}$ to be equipped with the row-by-column product; the identity matrix is denoted by Id. A map $\mu : A^* \to \mathbb{Q}^{n\times n}$ is a morphism if $\mu(\epsilon) = \mathrm{Id}$ and for any $w \in A^+$, $w = a_1a_2\cdots a_n$, $a_i \in A$, we have $\mu(w) = \mu(a_1)\mu(a_2)\cdots\mu(a_n)$. A $\mathbb{Q}$-set $S \in \mathbb{Q}^{A^*}$ is called *recognizable* or *representable* if there exists a positive integer n, λ, $\gamma \in \mathbb{Q}^n$ and a morphism $\mu : A^* \to \mathbb{Q}^{n\times n}$ such that for any $w \in A^*$

$$(S, w) = \lambda\mu(w)\gamma,$$

where λ and γ are to be considered row and column vectors, respectively. The triple (λ, μ, γ) is called a *linear representation* of S of dimension n or a $\mathbb{Q}$-*automaton* for S.

A *nondeterministic automaton* is a 5-tuple $M = (A, Q, E, I, F)$, where A is the input alphabet, Q is a finite set of *states*, $E \subseteq Q \times A \times Q$ is a set of *edges*, and $I, F \subseteq Q$ are, respectively, the sets of the *initial* and *final* states. Let $w = a_1a_2\cdots a_n \in A^*$. An *accepting path* for w is any sequence $\pi = (p_1, a_1, p_2)(p_2, a_2, p_3)\cdots(p_n, a_n, p_{n+1})$ with $p_1 \in I$, $p_{n+1} \in F$ and $(p_i, a_i, p_{i+1}) \in E$ for $1 \le i \le n$. The language accepted by M is the set $L(M)$ of all the words which have at least one accepting path; M is *unambiguous* if for any word $w \in L(M)$, there exists only one accepting path for w. We can associate with M a $\mathbb{Q}$-set $S_M \in \mathbb{Q}^{A^*}$, also called the *behavior* of M, defined as follows: for any $w \in A^*$, (S_M, w) is the number of different paths which are accepting for w. Let $Q = 1, 2, \ldots, n$ and let λ, $\gamma \in \mathbb{Q}^n$ be the characteristic vectors, respectively, of I and F; consider the morphism $\mu : A^* \to \mathbb{Q}^{n\times n}$, defined by $\mu(a)_{ij} = 1$ if $(i, a, j) \in E$ and $\mu(a)_{ij} = 0$ otherwise. Then we can easily prove (cf. [9, p. 137]) that for any $w \in A^*$

$$(S_M, w) = \lambda\mu(w)\gamma.$$

In particular, S_M is representable and, when M is unambiguous, S_M corresponds to the characteristic function of $L(M)$.

In general, a linear representation (λ, μ, γ) of dimension n can be regarded as an "automaton" whose set of states is $Q = \{1, 2, \ldots n\}$; initial and final states are defined as $\mathbb{Q}$-subsets of Q, while edges are a $\mathbb{Q}$-subset of $Q \times A \times Q$. Indeed, λ_i (resp. γ_i) represents the multiplicity of i as an initial state (resp. final state) and $\mu(a)_{i,j}$ represents the multiplicity of the edge (i, a, j). Probabilistic automata are particular $\mathbb{Q}$-automata [13]. A probabilistic automaton P can be represented by means of a linear representation (λ, μ, γ) with the following constraints: $\sum_{i=1}^n \lambda_i = 1$ and $\sum_{j=1}^n \mu(a)_{i,j} = 1$ for any $a \in A$ and $i \in \{1, 2, \ldots, n\}$; moreover, $0 \le \lambda_i \le 1$, $\gamma_i \in \{0, 1\}$, and $0 \le \mu(a)_{i,j} \le 1$ for any $a \in A$ and $i, j \in \{1, 2, \ldots, n\}$. Informally, λ_i represents the probability of i being an initial state, γ_i is 1 iff i is an accepting state, and $\mu(a)_{i,j}$ represents the probability of arriving in state j, starting from state i, and reading the input symbol a. Then the probability that P accepts when started with the distribution probability λ on Q and reading w is exactly $\lambda\mu(w)\gamma$. Finally, we shall need the following definitions.

DEFINITION 2.1. *For any string $u \in A^*$ and a $\mathbb{Q}$-set S, the $\mathbb{Q}$-set S_u is defined by* $(\forall w \in A^*)\,(S_u, w) = (S, uw)$.

DEFINITION 2.2. *For any set of strings $E \subseteq A^*$,*

$$S \equiv_E T \quad \text{iff } (\forall w \in E)\,(S, w) = (T, w).$$
$$S \equiv T \text{ stands for } S \equiv_{A^*} T.$$

We shall use an oracle for answering *multiplicity queries* for any string w, i.e., for providing the value of (S, w), where S is the target $\mathbb{Q}$-set.

3. Observation tables.

Based on previous work by Angluin on deterministic finite-state automata [3], we now introduce the concept of an observation table for a $\mathbb{Q}$-set S.

DEFINITION 3.1. *Let $S \in \mathbb{Q}^{A^*}$; an observation table is a triple $\mathcal{T} = (P, E, T)$, where P and E are sets of strings, P is prefix-closed, E is suffix-closed, and $T : (P \cup PA)E \to \mathbb{Q}$ gives observed values of S, i.e., for all strings $w \in (P \cup PA)E, T(w) = (S, w)$.*

Consequently, an observation table provides particular values for the target $\mathbb{Q}$-set S. Such values are obtained by means of multiplicity queries once the sets P and E are fixed. In Angluin's method for the exact identification of regular languages, the set P corresponds to states in an accepting DFA and the table contains some of the transitions. Here P determines a set $\{S_u | u \in P\}$ that will be useful in defining the target $\mathbb{Q}$-set S via linear dependencies. We then have the corresponding notions of closed and consistent observation tables.

DEFINITION 3.2. *An observation table (P, E, T) is closed iff $\forall u \in P$, $\forall a \in A$, there exists a coefficient $\alpha_v \in \mathbb{Q}$ for each $v \in P$ such that*

$$S_{ua} \equiv_E \sum_{v \in P} \alpha_v S_v. \tag{1}$$

DEFINITION 3.3. *An observation table (P, E, T) is consistent iff for any choice of coefficients $\beta_v \in \mathbb{Q}$ for each $v \in P$,*

$$\sum_{v \in P} \beta_v S_v \equiv_E 0 \Rightarrow (\forall a \in A) \sum_{v \in P} \beta_v S_{va} \equiv_E 0. \tag{2}$$

In Angluin's work (see also [2]), a natural notion of completeness was defined for P that required that all states in the target DFA have a representative in P. Here we have an analogous notion that requires that $\{S_u | u \in P\}$ be sufficient to establish all of the linear dependencies needed.

DEFINITION 3.4. *P is a complete set of strings for S iff $\forall u \in P$, $\forall a \in A$, there exists a coefficient $\lambda_v \in \mathbb{Q}$ for each $v \in P$ such that*

$$S_{ua} \equiv \sum_{v \in P} \lambda_v S_v. \tag{3}$$

When a table (P, E, T) is consistent and P is complete, the linear dependencies that are observed on E are valid for any string in A^*, as proved in the following.

THEOREM 3.5. *Let (P, E, T) be a consistent observation table, where P is a complete set of strings for S; then*

$$\sum_{v \in P} \beta_v S_v \equiv_E 0 \Rightarrow \sum_{v \in P} \beta_v S_v \equiv 0. \tag{4}$$

Proof. We shall prove the theorem by induction on $|y|$ by showing that for any $y \in A^*$,

$$\sum_{v \in P} \beta_v S_v \equiv_E 0 \Rightarrow \sum_{v \in P} \beta_v S_{vy} \equiv_E 0. \tag{5}$$

Base. $y = \epsilon$ and the thesis is trivially true.

Inductive step. Let $y = wb$ with $b \in A$.

By using the completeness of P a sufficient number of times, given $v \in P$, we may find coefficients $\lambda_{u,v}$, $u \in P$, such that

$$S_{vw} \equiv \sum_{u \in P} \lambda_{u,v} S_u. \tag{6}$$

Then for $x \in E$,

$$\begin{aligned} \sum_{v\in P} \beta_v (S_{vwb}, x) &= \sum_{v\in P} \beta_v (S_{vw}, bx) = \sum_{v \in P} \beta_v \sum_{u \in P} \lambda_{u,v} (S_u, bx) \\ &= \sum_{u \in P} \sum_{v \in P} \beta_v \lambda_{u,v} (S_u, bx) = \sum_{u \in P} \gamma_u (S_u, bx), \\ \text{where } \gamma_u &= \sum_{v \in P} \beta_v \lambda_{u,v}. \end{aligned} \tag{7}$$

By the inductive hypothesis, $\sum_{v \in P} \beta_v S_{vw} \equiv_E 0$; then, using (6),

$$\sum_{v \in P} \beta_v \sum_{u \in P} \lambda_{u,v} S_u \equiv \sum_{u \in P} \gamma_u S_u \equiv_E 0.$$

Again using the consistency of the table, we have

$$\sum_{u \in P} \gamma_u S_{ub} \equiv_E 0, \tag{8}$$

and, using this in (7), $\sum_{v \in P} \beta_v S_{vwb} \equiv_E 0$.

This completes the proof of (5). The theorem then follows from the fact that, since $\epsilon \in E$,

$$\left[(\forall y \in A^*) \sum_{v \in P} \beta_v S_{vy} \equiv_E 0 \right] \Rightarrow \sum_{v \in P} \beta_v S_v \equiv 0. \qquad \square \tag{9}$$

Consequently, the linear dependencies that show the table is closed are also valid in A^*.

COROLLARY 3.6. *Let (P, E, T) be a consistent observation table, where P is a complete set of strings for S; then for any $a \in A$,*

$$S_{ua} \equiv_E \sum_{v \in P} \lambda_v S_v \Rightarrow S_{ua} \equiv \sum_{v \in P} \lambda_v S_v. \tag{10}$$

Proof. Since P is complete, $S_{ua} \equiv \sum_{v \in P} \lambda'_v S_v$ for some $\lambda'_v \in \mathbb{Q}$. Therefore, if $S_{ua} \equiv_E \sum_{v \in P} \lambda_v S_v$, then $\sum_{v \in P} (\lambda'_v - \lambda_v) S_v \equiv_E 0$. By Theorem 3.5, $\sum_{v \in P} (\lambda'_v - \lambda_v) S_v \equiv 0$, and

$$S_{ua} \equiv \sum_{v \in P} \lambda'_v S_v \equiv \sum_{v \in P} \lambda_v S_v. \qquad \square$$

4. The learning algorithm. As is explained in what follows, there are stages in which the learning process builds a consistent and closed table. Here we only want to show how from such a table (P, E, T), we can guess a $\mathbb{Q}$-set $M(P, E, T)$ by basing its representation upon the existing linear dependencies:

- Let $P = \{u_1, \ldots, u_k\}$, with $u_1 = \epsilon$.

- For all $a \in A$, compute $\hat{\mu}(a)$ that satisfies

$$S_{u_i a} \equiv_E \sum_j \hat{\mu}(a)_{u_i,u_j} S_{u_j}. \tag{11}$$

Such a matrix exists because the table is closed. Moreover, the values of $\hat{\mu}(a)_{u_i,u_j}$ can be computed by solving the system of linear equations $(S_{u_i a}, v) = \sum_j x_{i,j}(S_{u_j}, v)$ with $v \in E$ in the unknowns $x_{i,j}$.

- Let $\hat{\lambda} = (1, 0, \ldots, 0)$ and $\hat{\gamma} = ((S_{u_1}, \epsilon), (S_{u_2}, \epsilon), \ldots, (S_{u_k}, \epsilon))$. The value of (S_{u_j}, ϵ) is found in the table since $u_j \in P$ and $\epsilon \in E$. Obviously, $\hat{\mu}(a)_{u_i,u_j}$ is the value at row i and column j of the matrix $\hat{\mu}(a)$. Let $\hat{\mu}(a_1 a_2 \cdots a_r) = \hat{\mu}(a_1)\hat{\mu}(a_2)\cdots\hat{\mu}(a_r)$, $a_i \in A$. Define the constructed $\mathbb{Q}$-set M with $(M, w) = \hat{\lambda}\hat{\mu}(w)\hat{\gamma}$.

THEOREM 4.1. *If P is a complete set of strings for S and (P, E, T) is a closed and consistent table, then $M(P, E, T) \equiv S$.*

Proof. Again, let $P = \{u_1, \ldots, u_k\}$ with $u_1 = \epsilon$. Since the table is closed, for any $a \in A$ $S_{u_i a} \equiv_E \sum_j \hat{\mu}(a)_{u_i,u_j} S_{u_j}$ and by Corollary 3.6, $S_{u_i a} \equiv \sum_j \hat{\mu}(a)_{u_i,u_j} S_{u_j}$. This may be easily generalized to any string t in A^*; by induction on $|t|$, we derive $S_{u_i t} \equiv \sum_j \hat{\mu}(t)_{u_i,u_j} S_{u_j}$. In fact let $t = sb$ with $b \in A$. By the induction hypothesis, we have $S_{u_i s} \equiv \sum_k \hat{\mu}(s)_{u_i,u_k} S_{u_k}$, which implies that $S_{u_i sa} \equiv \sum_k \hat{\mu}(s)_{u_i,u_k} S_{u_k a}$. On the other hand, from $S_{u_k a} \equiv \sum_j \hat{\mu}(a)_{u_k,u_j} S_{u_j}$, we derive $S_{u_i sa} \equiv \sum_k \sum_j \hat{\mu}(s)_{u_i,u_k} \hat{\mu}(a)_{u_k,u_j} S_{u_j} = \sum_j \hat{\mu}(sa)_{u_i,u_j} S_{u_j}$. Then

$$(S_{u_i}, t) = (S_{u_i t}, \epsilon) = \sum_j \hat{\mu}(t)_{u_i,u_j} (S_{u_j}, \epsilon). \tag{12}$$

Since $u_1 = \epsilon$, we have

$$(S, t) = \sum_j \hat{\mu}(t)_{u_1,u_j} (S_{u_j}, \epsilon) = \hat{\lambda}\hat{\mu}(t)\hat{\gamma} = (M, t). \qquad \square$$

We are now left with three problems: (i) closing a table; (ii) making a table consistent; (iii) making P complete. However, we will obtain completeness only indirectly and will return to it later.

4.1. Closing a table. Given a table (P, E, T) and $u \in P$, suppose that S_{ua} is linearly independent of $\{S_v | v \in P\}$ with respect to E in the sense that there are no coefficients $\lambda_{u,v}$ such that $S_{ua} \equiv_E \sum_{v \in P} \lambda_{u,v} S_v$. In this case, ua is added to P, and the table is again checked for closure.

This procedure must terminate; more precisely, if the correct $\mathbb{Q}$-set S is representable with $(S, x) = \lambda\mu(x)\gamma$, where $\lambda, \gamma \in \mathbb{Q}^n$ and $\mu : A^* \to \mathbb{Q}^{n \times n}$ is a morphism, then at most n strings can be added to P when closing the table.

In fact, it should be noted that when ua is added to P as indicated above, the dimension of $\{\lambda\mu(v) | v \in P\}$ as a subset of the vector space $\mathbb{Q}^n$ is increased by one. Otherwise, $\lambda\mu(ua)$ would be equal to $\sum_{v \in P} \beta_v \lambda\mu(v)$ for some coefficients β_v and

$$(S_{ua}, x) = (S, uax) = \lambda\mu(ua)\mu(x)\gamma = \sum \beta_v \lambda\mu(v)\mu(x)\gamma = \sum \beta_v (S_v, x),$$

i.e., S_{ua} would depend linearly on $\{S_v | v \in P\}$. Since the dimension of $\{\lambda\mu(v) | v \in P\}$ is at most n, we cannot close the table more than n times. The above discussion does not depend on E.

4.2. Making tables consistent. Given a table (P, E, T) and a symbol $a \in A$, consider the systems of linear equations

$$\text{(a)} \sum_{v \in P} \beta_v S_v \equiv_E 0, \qquad \text{(b)} \sum_{v \in P} \beta_v S_{va} \equiv_E 0$$

with β_v as unknowns. Check if every solution of system (a) is also a solution of system (b). In this case, the table is consistent. Otherwise, let β'_v, $v \in P$, be some solutions of (a) that are not solutions of (b) and let $x \in E$ such that $\sum_{v \in P} \beta_v (S_{va}, x) \neq 0$. Add ax to E. A method for checking whether every solution of (a) is also a solution of (b) is outlined in §4.4.

Suppose that S has a linear representation (λ, μ, γ) of dimension n; there cannot be more than n such additions to E because every time a new string ax is added, the dimension of $\{\mu(w)\gamma | w \in E\}$ is increased by one. In fact, if $\mu(ax)\gamma = \sum_{w \in E} \delta_w \mu(w)\gamma$, then

$$\begin{aligned}\sum_{v \in P} \beta_v (S_{va}, x) &= \sum_{v \in P} \beta_v \lambda \mu(v) \mu(ax) \gamma \\ &= \sum_{v \in P} \beta_v \lambda \mu(v) \sum_{w \in E} \delta_w \mu(w) \gamma \\ &= \sum_{w \in E} \delta_w \sum_{v \in P} \beta_v \lambda \mu(vw) \gamma \\ &= \sum_{w \in E} \delta_w \sum_{v \in P} \beta_v (S_v, w) = 0,\end{aligned}$$

i.e., ax would not have been added to E.

4.3. The algorithm. We may now describe the procedure for exactly identifying S from multiplicity queries and counterexamples.

```
$\mathcal{T} \leftarrow (\{\epsilon\},\{\epsilon\},T)$, where $(T,\epsilon) = (S,\epsilon)$.
Repeat
    • make the table closed and consistent ($P$ and $E$ are extended and the entries of $T$ are filled in by multiplicity queries); while closing the table, the hypothesized $\mathbb{Q}$-set $M$ is obtained;
    • ask for a counterexample $t$ to $M(P, E, T)$ by means of an equivalence query;
    • add $t$ and its prefixes to $P$
until $M$ is correct.
```

The main loop makes the table closed and consistent as described in §§4.1 and 4.2 and then constructs a guess M which is based on the observed linear dependencies. We shall now prove that if S has a linear representation (λ, μ, γ) of dimension n, after at most n equivalence queries, we will have a correct guess, i.e., $M \equiv S$. We need the following result.

LEMMA 4.2. *Let $u \in P$ and $t \in uA^*$, and suppose that for every $x \in A^*$ such that $t = uxz$, S_{ux} depends linearly on $\{S_v | v \in P\}$. Then for every prefix ux of t, after the table has been made closed and consistent during the execution of the algorithm, we have*

$$S_{ux} \equiv_E \sum_{v \in P} \hat{\mu}(x)_{u,v} S_v, \tag{13}$$

where $\hat{\mu} : A^* \to \mathbb{Q}^{k \times k}$ *is the morphism that corresponds to the observed linear dependencies as computed in* (11), *which is obtained when closing the table.*

Proof (by induction on $|x|$).

Base. $x = \epsilon$ and $S_u \equiv_E \sum_{v \in P} \hat{\mu}(\epsilon)_{u,v} S_v$ since $\hat{\mu}(\epsilon)$ is the identity matrix.

Inductive step. Let $x = yb$ and assume that $S_{uy} \equiv_E \sum_{v \in P} \hat{\mu}(y)_{u,v} S_v$.

Since uy is a prefix of t, S_{uy} must depend linearly on $\{S_v | v \in P\}$, i.e.,

$$S_{uy} \equiv \sum_{v \in P} \alpha_v S_v. \tag{14}$$

This along with the inductive hypothesis gives

$$\sum_{v \in P} (\alpha_v - \hat{\mu}(y)_{u,v}) S_v \equiv_E 0. \tag{15}$$

Then, since the table is consistent,

$$\sum_{v \in P} (\alpha_v - \hat{\mu}(y)_{u,v}) S_{vb} \equiv_E 0, \quad \text{i.e.,} \quad \sum_{v \in P} \alpha_v S_{vb} \equiv_E \sum_{v \in P} \hat{\mu}(y)_{u,v} S_{vb}. \tag{16}$$

Using (14) again, we have

$$S_{uyb} \equiv_E \sum_{v \in P} \hat{\mu}(y)_{u,v} S_{vb}. \tag{17}$$

Since $v \in P$, we may substitute $S_{vb} \equiv \sum_{w \in P} \hat{\mu}(b)_{v,w} S_w$ in the previous equation, which yields

$$S_{ux} \equiv S_{uyb} \equiv_E \sum_{v} \hat{\mu}(y)_{u,v} \sum_{w} \hat{\mu}(b)_{v,w} S_w \equiv \sum_{w} \hat{\mu}(yb)_{u,w} S_w. \qquad \square$$

THEOREM 4.3. *Let* $(S, t) \neq (M, t)$. *Then there is a prefix* t_0 *of* t *such that* S_{t_0} *is linearly independent of* $\{S_v | v \in P\}$.

Proof. If all prefixes t_0 of t would make S_{t_0} depend linearly on $\{S_v | v \in P\}$, then, by setting $u = \epsilon$ in the previous lemma and because $\epsilon \in E$, $(S, t) = (S_{\epsilon t}, \epsilon) = \sum_{v \in P} \hat{\mu}(t)_{\epsilon, v} (S_v, \epsilon) = \hat{\lambda} \hat{\mu}(t) \hat{\gamma}$. Then, however, $(M, t) = \hat{\lambda} \hat{\mu}(t) \hat{\gamma}$ would not be different from (S, t). $\square$

COROLLARY 4.4. *If* S *has a linear representation of dimension* n, *then after at most* n *iterations, the algorithm stops.*

Proof. Again suppose that the correct $\mathbb{Q}$-set S is representable with $(S, x) = \lambda \mu(x) \gamma$, where $\lambda, \gamma \in \mathbb{Q}^n$ and $\mu : A^* \to \mathbb{Q}^{n \times n}$. The algorithm will ask for a counterexample t and will add t and all of its prefixes to P. By Theorem 4.3, at each iteration, some prefix t_0 of the counterexample t is such that S_{t_0} is linearly independent of $\{S_u | u \in P\}$. However, $(S_{t_0}, x) = (S, t_0 x) = \lambda \mu(t_0) \mu(x) \gamma$ and for $u \in P$, $(S_u, x) = (S, ux) = \lambda \mu(u) \mu(x) \gamma$. Consequently, the dimension of $\{\lambda \mu(u) | u \in P\}$ is increased by one when t_0 is added to P. In fact, if by way of contradiction, $\lambda \mu(t_0) = \sum_{u \in P} \delta_u \lambda \mu(u)$ before t and its prefixes were added to P, then for all $x \in A^*$, $(S_{t_0}, x) = \lambda \mu(t_0) \mu(x) \gamma = \sum_{u \in P} \delta_u \lambda \mu(u) \mu(x) \gamma = \sum_{u \in P} \delta_u (S_u, x)$, i.e., S_{t_0} would depend linearly on $\{S_u | u \in P\}$. However, the dimension of $\{\lambda \mu(u) | u \in P\}$ cannot be larger than n. $\square$

4.4. Complexity analysis. All we need to do is reorganize some of the previous results and determine the complexity of computing the linear dependencies. Let n be the dimension of a linear representation of S. We have the following:

- The main loop in the algorithm is repeated at most n times (Corollary 4.4).
- $|\mathrm{E}| \leq n$ (§4.2).
- For the cardinality of P, the discussion is slightly more involved. From the discussions of §§4.1 and 4.3, we see that every time either (i) a string is added to P while closing the table or (ii) a counterexample is processed, the dimension of $\{\lambda\mu(v)|v \in P\}$ is increased by at least one. The worst case is when this always happens with the counterexamples and the main loop in the algorithm is repeated exactly n times, because the prefixes of the counterexamples also need to be added to P. If m is the maximum length of a counterexample, then $|P| \leq nm$.
- For every $a \in A$ and for every $u \in P$, the table needs to be closed. This amounts to solving at most knm systems of $|E|$ equations in $|P|$ unknowns—in the worst case, n simultaneous equations in nm unknowns. This can be done with Gauss's method with complexity $O(n^3m)$. Consequently, the complexity of closing the table is $O(kn^4m^2)$. We assume that the entries of the table are rational numbers of limited size; otherwise, the basic arithmetic operations involved in solving the systems of equations would be of arbitrary complexity. In general, it will be sufficient to require the entries of the table to be polynomial in n.
- Checking for consistency was described in §4.2 and requires the algorithm to confront the two systems of equations

$$\text{(a)} \sum_{v\in P} \beta_v S_v \equiv_E 0, \qquad \text{(b)} \sum_{v\in P} \beta_v S_{va} \equiv_E 0$$

with β_v as unknowns. The table is consistent if every solution of (a) is also a solution of (b). This is the same as checking whether the $|E|$ equations of system (b) do not add additional constraints, i.e., if the corresponding vectors of coefficients depend linearly on those of system (a). In the worst case, this operation requires checking whether there exists a solution for n systems of nm equations in n unknowns with complexity $O(n^4m)$. Since the operation must be performed for every $a \in A$, the overall complexity of checking for consistency is $O(kn^4m)$. The reason why we have at most n systems of nm equations in n unknowns is as follows. As explained above, in the worst case, $|E| = n$ and $|P| = nm$. Let $P = \{u_1, \ldots, u_{nm}\}$ and $E = \{e_1, \ldots, e_n\}$. Then system (a) is of the following type:

$$\beta_1(S_{u_1}, e_1) + \cdots + \beta_{nm}(S_{u_{nm}}, e_1) = 0$$
$$\cdots$$
$$\beta_1(S_{u_1}, e_n) + \cdots + \beta_{nm}(S_{u_{nm}}, e_n) = 0.$$

For each of the n equations of system (b) of the type

$$\beta_1(S_{u_1}, ae_i) + \cdots + \beta_{nm}(S_{u_{nm}}, ae_i) = 0,$$

we have to check whether the coefficients (S_{u_k}, ae_i) depend linearly on those of system (a), i.e., we have to check whether there exists a solution for the following system of nm equations in n unknowns:

$$\lambda_1(S_{u_1}, e_1) + \cdots + \lambda_n(S_{u_1}, e_n) = (S_{u_1}, ae_i)$$
$$\cdots$$
$$\lambda_1(S_{u_{nm}}, e_1) + \cdots + \lambda_n(S_{u_{nm}}, e_n) = (S_{u_{nm}}, ae_i).$$

- Filling the table is a task of lower complexity with respect to those considered above and does not influence the final result. In fact, in the worst case, the table is of size $|E||P| = n^2m$. Each multiplicity query for filling one entry of the table will be for a string of length at most $m + 2n + 1$. In fact, strings in P get as long as m when a counterexample is found, and one character may be added to them at most n times when closing the table. Strings in E are obtained by adding a one-character prefix to strings previously added to E, and this may be done at most n times. An extra character $a \in \Sigma$ is added at the time of the multiplicity query between a string in P and a string in E. This is done for all $a \in \Sigma$. The overall complexity of filling the table is $O(kn^2m(n + m))$.

Since the main loop is repeated at most n times, the overall complexity of the algorithm is $O(kn^5m^2)$. Together with the fact that when the main loop terminates, $M \equiv S$, this establishes our main result.

THEOREM 4.5. *Recognizable $\mathbb{Q}$-sets may be exactly identified in polynomial time from queries and counterexamples.*

This may be seen as a generalization of Angluin's result for finite-state automata [3]. It should be noted that Theorem 4.1 was the inspiration behind the algorithm but has not been used to obtain the above result. In particular, the completeness of P was not directly verified in the algorithm nor used in the proofs. However, when the algorithm terminates, the set P must be complete, as shown below.

5. Complete sets of strings. If $u \in P$, because tables are kept closed during the execution of the learning algorithm, for any $a \in A$,

$$S_{ua} \equiv_E \sum_{v \in P} \hat{\mu}(a)_{u,v} S_v. \tag{18}$$

In order to proceed with our discussion, we need an assumption that is quite obvious: if also $ua \in P$, then the linear dependencies for S_{ua} are chosen in the easiest way, i.e., with

$$\hat{\mu}(a)_{u,ua} = 1 \quad \text{and} \quad \hat{\mu}(a)_{u,v} = 0 \quad \text{for } v \neq ua. \tag{19}$$

Equation (19) is acceptable without loss of generality because it provides a hypothesized value $\hat{\mu}$ that satisfies (18). If $ua \notin P$, then any choice of $\hat{\mu}$ is acceptable as long as it satisfies (18). If this choice is made, the following holds.

LEMMA 5.1. *Let $u \in P$, and suppose that when $ua \in P$, $\hat{\mu}(a)_{u,v}$ is chosen as in equation* (19), *then for any $v \in P$, $\hat{\mu}(u)_{\epsilon,v} = 1$ if $v = u$ $\hat{\mu}(u)_{\epsilon,v} = 0$ otherwise.*

Proof (by induction on $|u|$).

Base. Since $\hat{\mu}$ is a morphism, $\hat{\mu}(\epsilon)$ must be the identity matrix, i.e., $\hat{\mu}(\epsilon)_{\epsilon,v} = 1$ when $v = \epsilon$ and is 0 elsewhere, which is what we need to prove.

Inductive step. Let $u = xa$, where $x \in P$ as $u \in P$ and P is prefix-closed.

$$\hat{\mu}(xa)_{\epsilon,v} = \sum_{w \in P} \hat{\mu}(x)_{\epsilon,w} \hat{\mu}(a)_{w,v}. \tag{20}$$

By the inductive hypothesis, $\hat{\mu}(x)_{\epsilon,w} = 0$ except when $w = x$, where it is equal to 1. Then the right-hand side of the previous equation reduces to

$$\hat{\mu}(x)_{\epsilon,x} \hat{\mu}(a)_{x,v} = 1 * \hat{\mu}(a)_{x,v}. \tag{21}$$

By (19) and because $xa = u \in P$, this is equal to 1 when $v = xa$ and is 0 elsewhere. □

LEMMA 5.2. *Under the same assumptions as in Lemma* 5.1, *for* $u \in P$, $\hat{\mu}(uw)_{\epsilon,v} = \hat{\mu}(w)_{u,v}$.

Proof.

$$\hat{\mu}(uw)_{\epsilon,v} = \sum_{z\in P} \hat{\mu}(u)_{\epsilon,z}\hat{\mu}(w)_{z,v}. \tag{22}$$

By the previous lemma, the right-hand side is equal to

$$\hat{\mu}(u)_{\epsilon,u}\hat{\mu}(w)_{u,v} = 1 * \hat{\mu}(w)_{u,v}. \quad \square \tag{23}$$

THEOREM 5.3. *Suppose that when* $ua \in P$, $\hat{\mu}(a)_{u,v}$ *is chosen as in equation* (19). *Then when* $M \equiv S$,

$$S_{ua} \equiv \sum_{v\in P} \hat{\mu}(a)_{u,v} S_v. \tag{24}$$

Consequently, when the guess M is correct, P must be complete.

Proof. We shall show that both sides of (24), when applied to any string $x \in A^*$, produce the same value.

$$\begin{aligned} (S_{ua}, x) &= (S, uax) = (M, uax) \\ &= \sum_{v\in P} \hat{\mu}(uax)_{\epsilon,v}(S_v, \epsilon) \\ &= \sum_{v\in P} \hat{\mu}(ax)_{u,v}(S_v, \epsilon) \quad \text{(by Lemma 5.2)}. \end{aligned} \tag{25}$$

$$\begin{aligned} \sum_{v\in P} \hat{\mu}(a)_{u,v}(S_v, x) &= \sum_{v\in P} \hat{\mu}(a)_{u,v}(S, vx) = \sum_{v\in P} \hat{\mu}(a)_{u,v}(M, vx) \\ &= \sum_{v\in P} \hat{\mu}(a)_{u,v} \sum_{w\in P} \hat{\mu}(vx)_{\epsilon,w}(S_w, \epsilon) \\ &= \sum_{v\in P} \hat{\mu}(a)_{u,v} \sum_{w\in P} \hat{\mu}(x)_{v,w}(S_w, \epsilon) \quad \text{(by Lemma 5.2)} \\ &= \sum_{w\in P}\sum_{v\in P} \hat{\mu}(a)_{u,v}\hat{\mu}(x)_{v,w}(S_w, \epsilon) = \sum_{w\in P} \hat{\mu}(ax)_{u,w}(S_w, \epsilon). \end{aligned}$$

This is equal to (25) after substituting v for w. □

Therefore, the completeness of P may be seen as a characterization of success when we learn that S, at least when $\hat{\mu}(ua)$ for $ua \in P$, is chosen as to verify (19). Moreover, it may be noted that if we start with a prefix-closed set P which is already complete, by virtue of Theorem 4.1, S may be exactly identified in polynomial time by means of multiplicity queries only. The algorithm is as follows:

1. $\mathcal{T} \leftarrow (P, \{\epsilon\}, T)$, where T is filled in with multiplicity queries.
2. Make the table $\mathcal{T}$ consistent, and fill in missing values with queries.
3. Output $M(\mathcal{T})$.

The table will certainly be closed since P is complete, and by Theorem 4.1, the final guess M must be correct. This result should be compared with [2], where a similar framework is described for finite-state automata: a regular language may be exactly identified in polynomial time using only membership queries, if we are given a complete set of representatives for Nerode's equivalence classes.

6. PAC-learnability and extensions. The learning method above also leads to some PAC-learnability results. If we do not require exact identification of the target $\mathbb{Q}$-set but are only interested in PAC-learnability, equivalence queries may be eliminated. Instead of asking an equivalence query, the algorithm will sample example strings and check whether the $\mathbb{Q}$-set learned at some stage is correct for these strings. The technique follows strictly from that used for DFAs in [3]. Consequently, $\mathbb{Q}$-automata are polynomially PAC-learnable when multiplicity queries are allowed. As a further consequence, nondeterministic automata may be PAC-predicted in polynomial time with multiplicity queries. It should be noted that negative results have been proved if only membership queries are available [4]. We do not have the proper PAC-learnability of NFAs with multiplicity queries because the representation that is learned is a $\mathbb{Q}$-automaton, not an NFA.

If a nondeterministic automaton is unambiguous, the corresponding $\mathbb{Q}$-set S will be such that for any string w, (S, w) is either 0 or 1. In this case, then, multiplicity queries reduce to ordinary membership queries. Now suppose that a regular language L is recognized by an unambiguous NFA M with a corresponding $\mathbb{Q}$-set S. This paper gives an algorithm for PAC-learning a representation of S in polynomial time with respect to the number of states of M if membership queries are allowed. In other words, regular languages are polynomially *predictable* using membership queries with respect to the representation of unambiguous nondeterministic automata. The importance of this lies in the fact that there are unambiguous NFAs such that the equivalent DFA has an exponentially larger number of states [16]. We then have a substantial improvement over previous results that established predictability with respect to a deterministic representation. It should be emphasized that the result only holds for *unambiguous* NFAs, and general NFAs are not predictable with membership queries, as shown in [4].

PAC-predictability is also established for probabilistic automata if multiplicity queries are allowed. As explained in §2, probabilistic automata may be represented by particular $\mathbb{Q}$-automata, and therefore they may be PAC-predicted with multiplicity queries. Again, we do not prove proper PAC-learnability because the learned representation is not a probabilistic automaton. In this case, multiplicity queries correspond to asking for the exact probability of accepting a given string. Previous results [17] required stronger types of queries. An interesting open problem is whether the present algorithm can be extended to the case where the oracle only provides an approximate probability of accepting some string. This would be more natural since one could think of estimating the probability by reading the string with the target probabilistic automaton several times.

Acknowledgments. We would like to thank the anonymous referees for the many remarks that helped us improve upon the initial version of this article.

REFERENCES

[1] D. Angluin, *On the complexity of minimum inference of regular sets*, Inform. and Control, 39 (1978), pp. 337–350.

[2] ———, *A note on the number of queries needed to identify regular languages*, Inform. and Control, 51 (1981), pp. 76–87.

[3] ———, *Learning regular sets from queries and counterexamples*, Inform. and Comput., 75 (1987), pp. 87–106.

[4] D. Angluin and M. Kharitonov, *When won't membership queries help?*, in Proc. 23rd Symposium on the Theory of Computing, Association for Computing Machinery, New York, 1991, pp. 444–454.

[5] D. Angluin, *Negative results for equivalence queries*, Mach. Learning, 5 (1990), pp. 121–150.

[6] F. Bergadano and D. Gunetti, *Inductive Logic Programming: From Machine Learning to Software Engineering*, MIT Press, Cambridge, MA, 1996.

[7] F. Bergadano and S. Varricchio, *Learning behaviours of automata from multiplicity and equivalence queries*, in Proc. 1994 Italian Conference on Algorithms and Complexity, Lecture Notes in Comput. Sci., 778, Springer-Verlag, Berlin, New York, Heidelberg, 1994, pp. 54–62.

[8] J. Berstel and C. Reutenauer, *Rational Series and Their Languages*, Springer-Verlag, Berlin, 1988.

[9] S. Eilenberg, *Automata, Languages and Machines*, Vol. A, Academic Press, New York, 1974.

[10] M. E. Gold, *Complexity of automaton identification from given data*, Inform. and Control, 37 (1978), pp. 302–320.

[11] T. Harju and J. Karhumaki, *Decidability of the multiplicity equivalence problem of multitape finite automata*, Proc. 22nd Symposium on the Theory of Computing, Association for Computing Machinery, New York, 1990, pp. 477–481.

[12] B. K. Natarajan, *Machine Learning: A Theoretical Approach*, Morgan Kaufmann, San Mateo, CA, 1991.

[13] A. Paz, *Introduction to Probabilistic Automata*, Academic Press, New York, 1991.

[14] L. Pitt and M. K. Warmuth, *The minimum consistent DFA problem cannot be approximated within any polynomial*, J. Assoc. Comput. Mach., 40 (1993), pp. 95–142.

[15] A. Salomaa and M. Soittola, *Automata Theoretic Aspects of Formal Power Series*, Springer-Verlag, New York, 1978.

[16] R. E. Stearns and H. B. Hunt, *On the equivalence and containment problems for unambiguous regular expressions, regular grammars, and finite automata*, SIAM J. Comput., 14 (1985), pp. 598–611.

[17] W. Tzeng, *Learning probabilistic automata and markov chains via queries*, Mach. Learning, 8 (1992), pp. 151–166.

[18] S. Varricchio, *On the decidability of the equivalence problem for partially commutative rational power series*, Theoret. Comput. Sci., 99 (1992), pp. 291–299.

SIAM J. COMPUT.
Vol. 25, No. 6, pp. 1281–1292, December 1996

008

PREFIX CODES: EQUIPROBABLE WORDS, UNEQUAL LETTER COSTS*

MORDECAI J. GOLIN† AND NEAL YOUNG‡

Abstract. We consider the following variant of Huffman coding in which the costs of the letters, rather than the probabilities of the words, are nonuniform: "Given an alphabet of r letters *of nonuniform length*, find a minimum-average-length prefix-free set of n codewords over the alphabet"; equivalently, "Find an optimal r-ary search tree with n leaves, where each leaf is accessed with equal probability but the cost to descend from a parent to its ith child depends on i." We show new structural properties of such codes, leading to an $O(n \log^2 r)$-time algorithm for finding them. This new algorithm is simpler and faster than the best previously known $O(nr \min\{\log n, r\})$-time algorithm, due to Perl, Garey, and Even [*J. Assoc. Comput. Mach.*, 22 (1975), pp. 202–214].

Key words. algorithms, Huffman codes, prefix codes, trees

AMS subject classification. 68Q25

1. Introduction.

The well-known Huffman coding problem [3] is the following: given a sequence of access probabilities $\langle p_1, p_2, \ldots, p_n \rangle$, construct a binary prefix code $\langle w_1, w_2, \ldots, w_n \rangle$ minimizing the expected length $\sum_i p_i \cdot \text{length}(w_i)$. A *binary prefix code* is a set of binary strings, none of which is a prefix of another.

A natural generalization of the problem is to allow the words of the code to be strings over an arbitrary alphabet of $r \geq 2$ letters and to allow each letter to have an arbitrary nonnegative length. The length of a codeword is then the sum of the lengths of its letters. For instance, the "dots and dashes" of Morse code are a variable-length alphabet with length corresponding to transmission time. (See Figure 1.) This generalization of Huffman coding to a variable-length alphabet has been considered by many authors, including Altenkamp and Mehlhorn [1] and Karp [5]. Apparently, no polynomial-time algorithm for it is known, nor is it known to be NP-hard.

A prefix code in which the codewords $\langle w_1, w_2, \ldots, w_n \rangle$ are in alphabetical order is called *alphabetic* [1]. In this case, the underlying tree represents an r-ary *search tree*. The length of the ith letter corresponds to the time required to descend from a node into its ith subtree. This time is often a function of i in search-tree algorithms, for instance, when the subtree to descend into is chosen by sequential search. An optimal alphabetic code thus corresponds to a minimum-expected-cost search tree.

In this paper, we consider the special case in which the codewords occur with equal probability, i.e., each p_i equals $1/n$. With this restriction, the alphabetic and nonalphabetic problems are equivalent. The problem may be viewed as a variant of Huffman coding in which the lengths of the letters, rather than the codeword probabilities, are nonuniform. Alternatively, it may viewed as the problem of finding an optimal r-ary search tree, where the search queries are uniformly distributed but the time to descend from a parent to its ith child depends on i. For the complexity results stated in this paper, the algorithms return a tree representing an optimal code.

* Received by the editors May 25, 1994; accepted for publication (in revised form) March 1, 1995.

† Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong (golin@cs.ust.hk). The research of this author was partially supported by HK RGC Competitive Research grant HKUST 181/93E.

‡ UMIACS, University of Maryland, College Park, MD 20742. Current address: Department of Computer Science, Dartmouth College, Hanover, NH 03755-3510 (neal.young@dartmouth.edu). The research of this author was partially supported by NSF grants CCR-8906949 and CCR-9111348.

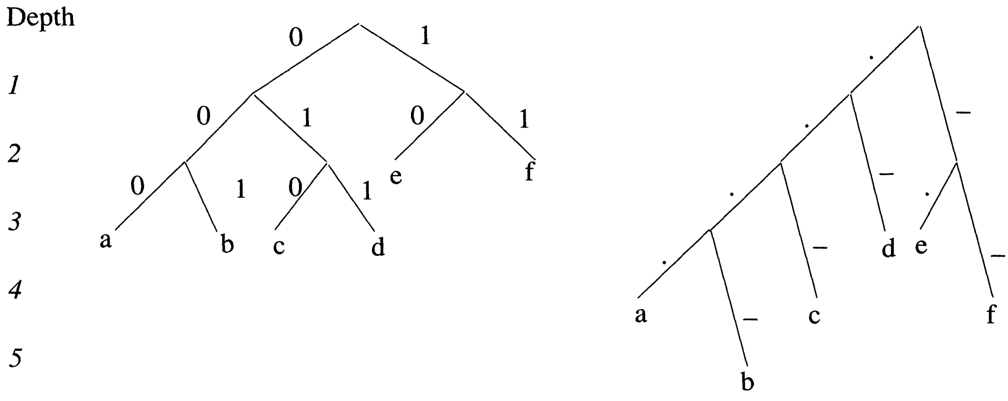


FIG. 1. *Two trees for the six symbols a, b, c, d, e, and f, each occurring with probability* 1/6. *The tree on the left is the optimal tree that uses the alphabet* $\{0, 1\}$, *with length*$(0) =$ *length*$(1) = 1$, *while the tree on the right is for the alphabet* $\{., \_\}$ *with length*$(.) = 1$ *and length*$(\_) = 2$. *The corresponding sets of codewords are*

$$a = 000, \quad b = 001, \quad c = 011, \quad d = 011, \quad e = 10, \quad f = 11$$

and

$$a = ..., \quad b = ...\_, \quad c = ...\_, \quad d = .\_, \quad e = \_., \quad f = \_\_ \ .$$

In 1989, Kapoor and Reingold [4] described a simple $O(n)$-time algorithm for the binary case $r = 2$. In 1975, Perl, Garey, and Even [7] gave an $O(rn \min\{r, \log n\})$-time algorithm (though due to a typographical error, their abstract incorrectly claims an $O(rn)$-time algorithm). In the same year, Cot [2] described an $O(r^2 n)$-time algorithm. In 1971, Varn [8] gave an algorithm without analyzing its complexity. It appears Varn's algorithm requires $\Omega(rn)$ time.

In this paper, we describe an $O(n \log^2 r)$-time algorithm based on new insights into the structure of optimal trees. In §2, we define *shallow* and *proper* trees and prove that some proper shallow tree is optimal. In §3, we develop the algorithm, which efficiently constructs all proper shallow trees and returns one representing an optimal prefix code.

2. Shallow trees. Fix an instance of the problem, given by the respective lengths $\langle c_1 \leq c_2 \leq \cdots \leq c_r \rangle$ of the r letters in the alphabet and the number n of (equiprobable and prefix-free) codewords required. We assume the standard tree representation of prefix codes, as described in the following definition.

DEFINITION 2.1. *The* infinite r-ary tree *is the infinite, rooted, r-ary tree. Each tree edge has a length and a label—an edge going from a node to its ith child has length c_i and is labeled with the ith letter in the alphabet.*

A node *is a node of the infinite r-ary tree.* The finite words over the alphabet of r letters correspond to the nodes. The labels along the path from the root to any node spell the corresponding word and the length of the path is the length of this word. A prefix code corresponds to a set of nodes none of which is a descendant of another. (See Figure 1.)

DEFINITION 2.2. *A* tree *is any subtree T of the infinite r-ary tree containing the root. In any tree, n of the leaves will be identified as* terminals; *their corresponding words form a prefix code. The remaining nodes in the tree are referred to as* nonterminals.

Given a node u, the notation $child_i(u)$ denotes u's ith child; $depth(u)$ denotes the depth (the length of the corresponding codeword); $parent(u)$ denotes the parent.

The cost $c(T)$ *of such a tree is the sum of the depths of the terminals—also called the* external weighted path length *of the tree.*

A proper *tree is a tree in which every nonterminal has at least two children.*

The goal is to find an optimal tree with n terminals. It is easy to see that some optimal tree is proper; thus we restrict our attention to proper trees.

Our basic tool for understanding the structure of optimal trees is a swapping argument. For example, in any proper optimal tree, no nonterminal is deeper than any terminal. Otherwise, the terminal and the subtree rooted at the nonterminal could be swapped, decreasing the average depth of the terminals.

We use a swapping argument to prove that an optimal proper tree has the following form for some m. The nonterminals are the m shallowest (i.e., least-depth) nodes of the infinite tree, while the terminals are the n shallowest available children of these nodes in the infinite tree. We call such a tree *shallow*; here is the precise definition.

DEFINITION 2.3. *A tree T is* shallow *provided that*

(i) *for any nonterminal $u \in T$ and any node w (not necessarily in T) that is not a nonterminal, $depth(u) \leq depth(w)$ and*

(ii) *for any terminal $u \in T$ and any node w that is not in T but is a child of a nonterminal, $depth(u) \leq depth(w)$.*

Note that a nonterminal of an (improper) shallow tree might have no children in the tree. This is why we refer to "terminal" and "nonterminal" nodes in place of the more common "internal nodes" and "leaves."

As a simple example, consider the basic binary tree; $r = 2$, $c_1 = c_2 = 1$. A proper binary tree T will be shallow if and only if there is some depth l such that (a) every node u in the infinite tree with $depth(u) < l$ is a nonterminal in T and (b) all terminals of T are on levels l and $l + 1$. Conditions (a) and (b) are necessary and sufficient conditions for T to have minimum external path length among all binary trees with the same number of leaves; see, e.g., [6, §5.3.1]. So, a binary tree has minimum external path length for its number of leaves if and only if it is shallow. For example, the binary tree on the left-hand side of Figure 1 has minimum external path length among all trees with six leaves because it fulfills conditions (a) and (b) with $l = 2$. As we will see later, though, for most values of r and c_i, shallowness alone does not imply optimality. However, if a shallow tree has the right number of nonterminals, then it is optimal.

LEMMA 2.4. *Let m^* be the minimum number of nonterminals in any optimal tree. Then any shallow tree with m^* nonterminals is optimal and proper.*

Proof. Fix a shallow tree T with m^* nonterminals. We will show the existence of an optimal tree with the same nonterminals as T. Since T is shallow, by property (ii), this will imply that T is optimal. By the choice of m^*, T is also proper (otherwise there would be an optimal proper tree with fewer nonterminals).

It remains to show the existence of an optimal tree with the same nonterminals as T. Let T^* be an optimal (and therefore proper) tree with m^* nonterminals. Let N and N^* be the sets of nonterminals of T and T^*, respectively. If $N = N^*$, we are done. Otherwise, let u be a minimum-depth node in $N - N^*$, so that u's parent is in N^*. Let u^* be a node in $N^* - N$. Note that, since T is shallow, $depth(u^*) \geq depth(u)$ but that, in T^*, u^* is a nonterminal (with at least two terminal descendants) while u is either a terminal or not present.

In T^*, swap the subtrees rooted at u and u^*. Specifically, make u a nonterminal

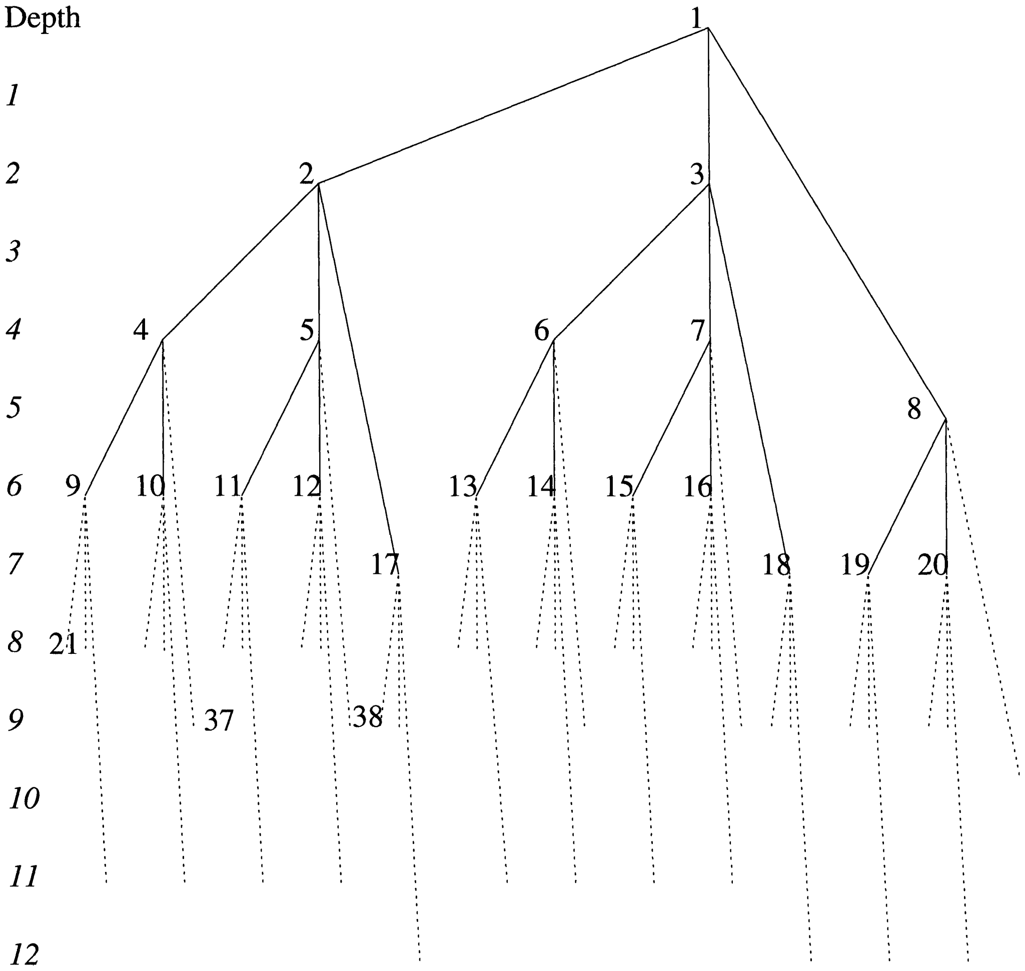


FIG. 2. *The top of a labeled infinite tree with $r = 3$, $c_1 = 2$, $c_2 = 2$, and $c_3 = 5$.*

and, for each descendant v^* of u^*, delete it and add the corresponding descendant v of u. If v^* was a terminal, make v a terminal; otherwise, make v a nonterminal. If u was a terminal, make u^* a terminal; otherwise, delete u^*. Call the resulting tree T'.

From $depth(u^*) \geq depth(u)$, it follows that $c(T') \leq c(T^*)$. Thus T' is also optimal. Note that T' shares one more nonterminal with T than does T^*. Thus repeated swapping produces an optimal tree with the same nonterminals as T. □

Note that $m^* \geq (n-1)/(r-1)$ since each node has degree at most r.

COROLLARY 2.5. *Let $m_{\min} = \lceil (n-1)/(r-1) \rceil$. Let $\langle T_{m_{\min}}, T_{m_{\min}+1}, T_{m_{\min}+2}, \ldots \rangle$ be any sequence of shallow trees such that for each m, T_m has m nonterminals. Then one of the T_m is proper and optimal.*

The algorithm generates a sequence of shallow trees as above and returns the one which has minimum cost. The lemma guarantees that this tree will be optimal. The rest of the paper is devoted to examining the properties of shallow trees which enable the enumeration of the proper shallow trees in $O(n \log^2 r)$ time.

2.1. Defining the trees.

Ordering the nodes. Label the nodes of the infinite tree as $1, 2, 3, \ldots$ in order of increasing depth. Break ties arbitrarily, except that if two nodes u and w are of equal depth, both are ith children of their respective parents, and $\mathrm{parent}(u) < \mathrm{parent}(w)$, then let $u < w$ (this is needed for Lemma 3.2). For the sake of notation, identify each node with its label so that 1 is the root, 2 is a minimum-depth child of the root, etc. Figure 2 illustrates the top section of such a labeling for $r = 3$, $c_1 = 2$, $c_2 = 2$, and $c_3 = 5$. These values of r and c_j are the ones we use in all later examples.

DEFINITION 2.6. *For each $m \geq m_{\min}$, define T_m to be the tree whose nonterminals are $\{1, \ldots, m\}$ and whose terminals are the minimum n nodes among the children of $\{1, \ldots, m\}$ in $\{m+1, m+2, \ldots\}$.*

Thus T_m is the "shallowest" tree with m nonterminals with respect to the ordering of the nodes. Since the ordering of the nodes respects depth, each T_m is shallow. Figure 3 presents T_5, T_6, T_7, and T_8 for $n = 10$ using the labeling of Figure 2.

2.2. Relation of successive trees.

Next, we turn our attention to the relation of T_{m+1} to T_m.

LEMMA 2.7. *For $m \geq m_{\min}$, the new nonterminal (node $m+1$) in T_{m+1} is the minimum terminal of T_m.*

Proof. The parent of $m+1$ is in $\{1, \ldots, m\}$, so $m+1$ is the minimum child of $\{1, \ldots, m\}$ in $\{m+1, m+2, \ldots\}$. The result follows from the definition of T_m. □

LEMMA 2.8. *For $m \geq m_{\min}$, provided the new nonterminal (node $m+1$) in T_{m+1} has at least one child, each terminal of T_{m+1} is either a child of $m+1$ or a terminal of T_m.*

Proof. Let node $m+1$ have d children in T_{m+1}. Let $\mathcal{C}$ denote the set of children of nodes $\{1, \ldots, m\}$ in $\{m+1, m+2, \ldots\}$. The terminals of tree T_{m+1} consist of the minimum d children of node $m+1$ together with the minimum $n-d$ nodes in $\mathcal{C} - \{m+1\}$. These $n-d$ nodes together with node $m+1$ (the minimum node in $\mathcal{C}$) are the $n-d+1$ minimum nodes in $\mathcal{C}$. If $d \geq 1$, then by the definition of T_m, each such node is a terminal in T_m. □

The main significance of Lemmas 2.7 and 2.8 is that they will allow an efficient construction of T_{m+1}. Moreover, they imply that if T_m is not proper, then neither is any subsequent tree.

LEMMA 2.9. *One of the trees $\langle T_{m_{\min}}, T_{m_{\min}+1}, \ldots, T_{m_{\max}} \rangle$ is optimal and proper, where $m_{\max} = \min\{m : T_{m+1}$ is improper$\}$.*

Proof. By Lemma 2.8, if T_m is improper, then so is T_{m+1}—either node $m+1$ has no children in T_{m+1} or the nonterminal in T_m that had less than two children also has less than two children in T_{m+1}. Hence, for each $m > m_{\max}$, tree T_m is improper. Thus Corollary 2.5 implies that one of the trees $\langle T_{m_{\min}}, T_{m_{\min}+1}, \ldots, T_{m_{\max}} \rangle$ is proper and optimal. □

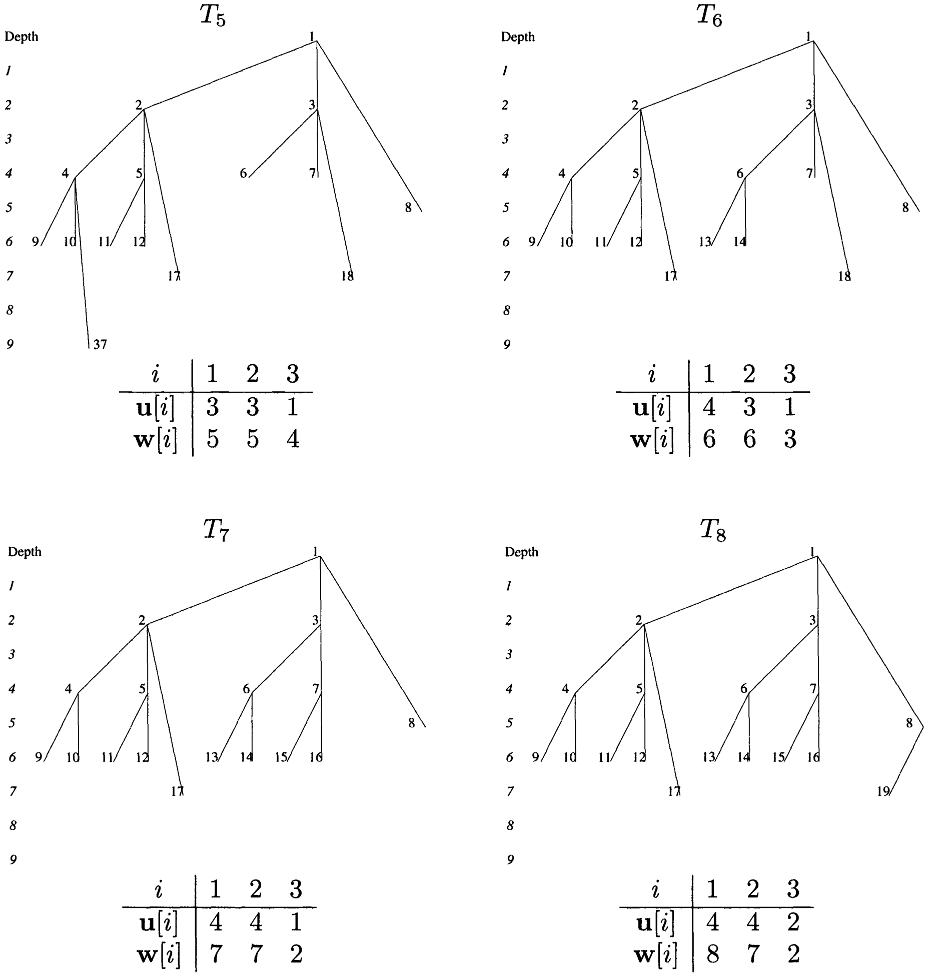
For $n = 10$, $m_{\min} = \lceil \frac{10-1}{3-1} \rceil = 5$ and (as shown in Figure 3) T_8 is improper. The lemma then implies that one of T_5, T_6, or T_7 must have minimum external path length. Calculation shows that T_6 with $c(T_6) = 59$ is the optimal one.

3. Computing the trees.

The algorithm uses the following two operations to compute the trees.

To SPROUT a tree is to make its minimum terminal a nonterminal and to add the minimum child of this nonterminal as a terminal.

To LEVEL a tree is to add c children of the maximum nonterminal to the tree as terminals and to remove the c largest terminals in the tree. The c children are the



T_5

i	1	2	3
u[i]	3	3	1
w[i]	5	5	4

T_6

i	1	2	3
u[i]	4	3	1
w[i]	6	6	3

T_7

i	1	2	3
u[i]	4	4	1
w[i]	7	7	2

T_8

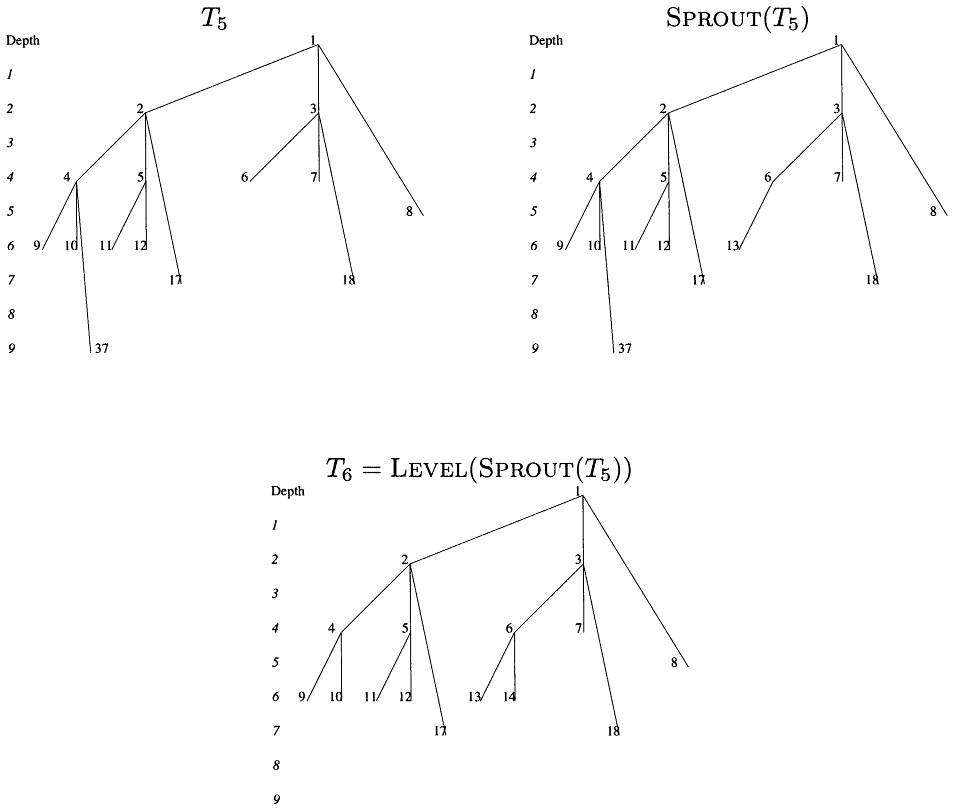
i	1	2	3
u[i]	4	4	2
w[i]	8	7	2

FIG. 3. *The trees T_5, T_6, T_7, and T_8 for $r = 3$, $c_1 = 2$, $c_2 = 2$, $c_3 = 5$, and $n = 10$. The node numbering is that of the previous figure. Calculating the external path lengths, we find that $c(T_5) = 60$, $c(T_6) = 59$, $c(T_7) = 60$, and $c(T_8) = 62$.*

minimum c children not yet in the tree, where c is maximum such that all children added are less than all terminals deleted.

The algorithm computes the initial tree $T_{m_{\min}}$ and then repeatedly SPROUTs and LEVELs to obtain successive trees until the tree so obtained is not proper. Lemmas 2.7 and 2.8 imply that, as long as node $m+1$ has at least one child in T_{m+1} (it will if T_{m+1} is proper), SPROUTing and LEVELing T_m yields T_{m+1}. Figure 4 illustrates this operation.

OBSERVATION 3.1. *Let $m = m_{\max}$. If node $m+1$ has at least one child in T_{m+1} then* SPROUT*ing and* LEVEL*ing T_m yields tree T_{m+1}. If node $m+1$ has no children in T_{m+1}, then the maximum terminal in T_m is less than the minimum child of node $m+1$ and* SPROUT*ing and* LEVEL*ing T_m yields a tree in which nonterminal $m+1$ has one child. Hence the algorithm always correctly identifies $T_{m_{\max}}$ and terminates*



T_5

$\textsc{Sprout}(T_5)$

$T_6 = \textsc{Level}(\textsc{Sprout}(T_5))$

FIG. 4. SPROUT*ing and* LEVEL*ing* T_5 *yields* T_6.

correctly, having considered all relevant trees.

To SPROUT requires identification and conversion of the minimum terminal of the current tree, whereas to LEVEL requires identification and replacement of (no more than r) maximum terminals by children of the new nonterminal. One could identify the maximum and minimum terminals in $O(\log n)$ time by storing all terminals in two standard priority queues (one to detect the minimum, the other to detect the maximum). At most r terminals would be replaced in computing each tree and, because $m_{\max} \leq n-1$, only $O(n)$ trees would be computed. This approach yields an $O(rn \log n)$-time algorithm.

By a more careful use of the structure of the trees, we improve this in two ways. First, we give an amortized analysis showing that in total, only $O(n \log r)$, rather than $O(rn)$, terminals are replaced. Second, we show how to reduce the number of nonterminals in each priority queue to at most r. This yields an $O(n \log^2 r)$-time algorithm.

Both improvements follow from the tie-breaking condition on the ordering of the nodes, which guarantees that T_m must have the following structure.

LEMMA 3.2. *In any T_m, if u and w are nonterminals with $u < w$ and the ith child of w is in the tree, then so is the ith child of u. If the ith child of w is a nonterminal, then so is the ith child of u.*

Proof. The proof is straightforward from the definition of T_m and the condition on breaking ties in ordering the nodes (in §2.1). □

COROLLARY 3.3. *Node m has a minimum number of children among all nonterminals in T_m.*

3.1. Only $O(n \log r)$ replacements total.

The number of terminals replaced while obtaining T_m from T_{m-1} is at most the number of children of nonterminal m in T_m. Although this might be r for many m, the sum of the numbers of children is $O(n \log r)$.

LEMMA 3.4. *Let d_m be the number of children of nonterminal m in tree T_m. Then $\sum_m d_m$ is $O(n \log r)$.*

Proof. By Corollary 3.3, within T_m, node m has the fewest children. The total number of children of the m nonterminals is $m+n-1$. Thus d_m is at most the average $(m+n-1)/m = 1 + (n-1)(1/m)$.

$$\begin{aligned}\sum_{m=m_{\min}}^{m_{\max}} d_m &\le (m_{\max} - m_{\min} + 1) + (n-1) \sum_{m=m_{\min}}^{m_{\max}} 1/m \\ &= O(m_{\max} - m_{\min} + n \log(m_{\max}/m_{\min})).\end{aligned}$$

The result follows from $m_{\min} = \lceil \frac{n-1}{r-1} \rceil$ and $m_{\max} \le n-1$. □

3.2. Limiting the relevant terminals.

To reduce the number of terminals that must be considered in finding the minimum and maximum terminals, we partition the terminals into r groups. The ith group consists of the terminals that are ith children $(i = 1, \ldots, r)$.

LEMMA 3.5. *In any T_m, for any i, the set of nonterminals whose ith children are terminals is of the form $\{u_i, u_i + 1, \ldots, w_i\}$ for some u_i and w_i. The minimum among terminals that are ith children is $\text{child}_i(u_i)$ (the ith child of u_i). The maximum among these terminals is $\text{child}_i(w_i)$.*

Proof. This is a straightforward consequence of Lemma 3.2. □

Figure 3 presents u_i and w_i for the trees T_5, T_6, T_7, and T_8 when $n = 10$.

This lemma implies that the minimum terminal in T_m is the minimum among $\{\text{child}_i(u_i) : i = 1, \ldots, r\}$. Our algorithm finds the minimum terminal in T by maintaining these r particular children (rather than all n terminals) in a priority queue. This reduces the cost of finding the minimum from $O(\log n)$ to $O(\log r)$. Similarly, the algorithm finds the maximum terminal in $O(\log r)$ time by maintaining $\{\text{child}_i(w_i) : i = 1, \ldots, r\}$ in an additional priority queue.

OBSERVATION 3.6.[1] *As an aside, one can prove using Lemma 3.5 that, for any m such that $m_{\min} < m < m_{\max}$, $c(T_{m+1}) - c(T_m) \ge c(T_m) - c(T_{m-1})$. That is, the sequence of tree costs is unimodal. To prove this, consider building T_{m+1} from T_m. SPROUTing increases the cost by c_1; LEVELing decreases the cost with each swap. For each swap in building T_{m+1} from T_m, one can show there was a corresponding swap in building T_m from T_{m-1} and that the decrease in cost (from T_m to T_{m+1}) due to the former is bounded by the decrease in cost (from T_{m-1} to T_m) due to the latter. Thus, in practice, the algorithm could be modified to stop and return T_{m-1} when $c(T_m) \ge c(T_{m-1})$.*

[1] This observation is due to R. Fleischer.

3.3. The algorithm in detail. The full algorithm has two distinct phases. The first phase constructs the base tree $T_{m_{\min}}$. The second phase starts with $T_{m_{\min}}$ and, by SPROUTing and LEVELing, iteratively constructs the sequence of shallow trees

$$\langle T_{m_{\min}}, T_{m_{\min}+1}, T_{m_{\min}+2}, \ldots, T_{m_{\max}} \rangle$$

and returns one which has smallest external path length. $T_{m_{\max}}$ is the last proper tree in the sequence, i.e., $T_{m_{\max}+1}$ is improper. Lemma 2.9 guarantees that the algorithm returns an optimal tree. We now describe how to implement the first part of the algorithm in $O(n \log r)$ time and the second in $O(n \log^2 r)$ time; the full algorithm therefore runs in $O(n \log^2 r)$ time.

The skeleton of the final algorithm is shown in Figure 5. Procedure CREATE-$T_{m_{\min}}$ creates tree $T_{m_{\min}}$, the variable **C** contains the external path length of current tree T_m, and **mDeg** contains the number of children of node m in tree T_m. As presented, the algorithm computes only the cost of an optimal tree. It can easily be modified to compute the actual tree. Note that to check that the current tree T_m is proper, by Observation 3.1 and Corollary 3.3, it suffices to check that nonterminal m has at least two children.

```
COMPUTE-TREES($\langle c_1, c_2, \ldots, c_r \rangle, n$)
    CREATE-$T_{m_{\min}}$
    WHILE ($\mathbf{mDeg} \geq 2$) DO
            —Compute $T_{\mathbf{m}+1}$ from $T_{\mathbf{m}}$—
            SPROUT($T$)
            LEVEL($T$)
            $\mathbf{C}_{\min} \leftarrow \min\{\mathbf{C}, \mathbf{C}_{\min}\}$
    RETURN $\mathbf{C}_{\min}$
```

FIG. 5. *Algorithm to find an optimal variable-length prefix code.*

The routines SPROUT and LEVEL are shown in Figure 6.

Recall that the nodes of the infinite tree are labeled in order of increasing depth with ties broken arbitrarily except for the requirement that if u and v are both of equal depth and both are ith children of their respective parents, then $u < v$ if and only if $\text{parent}(u) < \text{parent}(v)$. Depending upon $c_1, c_2, \ldots, c_r$, there may be many such labelings. The algorithm we present breaks ties lexicographically—suppose u and v have the same depth and let $u = \text{child}_i(u')$ and $v = \text{child}_j(v')$; then $u < v$ if and only if $u' < v'$ (or $u' = v'$ and $i < j$). Figure 2 illustrates this labeling for $r = 3$, $c_1 = 2$, $c_2 = 2$, and $c_3 = 5$. The sequence of shallow trees is fully determined by this labelling. Figure 3 illustrates the shallow trees with 10 nonterminals for these r and c values.

The algorithm represents the current tree T_m with the following data structures:

N is the number of terminals.

m is the number of nonterminals. It is also the rank of the maximum nonterminal.

C is the sum of the depths of the terminals.

mDeg is the number of children of nonterminal **m**.

$\mathbf{D}[u]$ is the depth of each nonterminal u.

$\mathbf{u}[i]$ is the rank of the minimum nonterminal (if any) whose ith child is a terminal $(1 \leq i \leq r)$.

$\mathbf{w}[i]$ is the rank of the maximum nonterminal (if any) whose ith child is a terminal $(1 \leq i \leq r)$. If no nonterminal has a terminal ith child, then $\mathbf{u}[i] > \mathbf{w}[i]$.

```
SPROUT($T$)
    —Make the minimum terminal a nonterminal—
 1.   $\mathbf{m} \leftarrow \mathbf{m} + 1$;
 2.   Let $\text{child}_i(\mathbf{u}[i])$ be the minimum terminal in low-queue.
 3.   $\mathbf{D}[\mathbf{m}] \leftarrow \mathbf{D}[\mathbf{u}[i]] + c_i$; $\mathbf{u}[i] \leftarrow \mathbf{u}[i] + 1$; UPDATE-QS($T, i$)
 4.   $\mathbf{C} \leftarrow \mathbf{C} - \mathbf{D}[\mathbf{m}]$; $\mathbf{mDeg} \leftarrow 0$;
    —Add smallest child as a terminal—
 5.   ADD-TERMINAL($T$)

LEVEL($T$)
 1. WHILE ($\mathbf{mDeg} < r$ and $\text{child}_{\mathbf{mDeg}+1}(\mathbf{m})$ is less than
             the max. terminal $\text{child}_i(\mathbf{w}[i])$ in high-queue) DO
 2.         ADD-TERMINAL($T$)
            —Delete the maximum terminal—
 3.         $\mathbf{C} \leftarrow \mathbf{C} - (\mathbf{D}[\mathbf{w}[i]] + c_i)$
 4.         $\mathbf{w}[i] \leftarrow \mathbf{w}[i] - 1$; UPDATE-QS($T, i$)

ADD-TERMINAL($T$)
 1. $\mathbf{mDeg} \leftarrow \mathbf{mDeg} + 1$; $\mathbf{C} \leftarrow \mathbf{C} + \mathbf{D}[\mathbf{m}] + c_{\mathbf{mDeg}}$;
 2. $\mathbf{w}[\mathbf{mDeg}] \leftarrow \mathbf{m}$; UPDATE-QS($T, \mathbf{mDeg}$)
```

FIG. 6. *Operations* SPROUT *and* LEVEL.

low-queue is a priority queue for finding the minimum terminal. It contains $\{\text{child}_i(\mathbf{u}[i]) : \mathbf{u}[i] \le \mathbf{w}[i]\}$.

high-queue is a priority queue for finding the maximum terminal. It contains $\{\text{child}_i(\mathbf{w}[i]) : \mathbf{u}[i] \le \mathbf{w}[i]\}$.

For an example, refer back to Figure 3. Tree T_6 has

$$N = 10, \qquad C = 59, \qquad \mathbf{mDeg} = 2,$$

$$D[1] = 0, \qquad D[2] = 2, \qquad D[3] = 3, \qquad D[4] = 4, \qquad D[5] = 4, \qquad D[6] = 4,$$

$$\mathbf{u}[1] = 4, \qquad \mathbf{u}[2] = 3, \qquad \mathbf{u}[3] = 1, \qquad \mathbf{w}[1] = 6, \qquad \mathbf{w}[2] = 6, \qquad \mathbf{w}[3] = 3,$$

$$\textbf{low-queue} = \{\text{child}_1(4), \text{child}_2(3), \text{child}_3(1)\},$$

$$\textbf{high-queue} = \{\text{child}_1(6), \text{child}_2(6), \text{child}_3(3)\}.$$

The priority queues are maintained as follows. In general, a terminal in T_m can have rank (label) arbitrarily larger than m. The algorithm explicitly maintains the ranks and depths of the m nonterminals in the current tree; the algorithm compares the ranks of terminals in the priority queues via the ranks and depths of their (nonterminal) parents. When $\mathbf{u}[i]$ or $\mathbf{w}[i]$ changes to reflect a new current tree, the queues are updated by the following routine:

```
UPDATE-QS($T, i$)
 1. IF ($\mathbf{u}[i] \le \mathbf{w}[i]$) THEN
 2.    Update $\text{child}_i(\mathbf{u}[i])$ in low-queue and $\text{child}_i(\mathbf{w}[i])$ in high-queue
       to maintain the queues' invariants.
 3. ELSE Delete both nodes from their respective queues.
```

Line 2 replaces the old $\text{child}_i(\mathbf{u}[i])$ in **low-queue** ($\text{child}_i(\mathbf{w}[i])$ in **high-queue**) by the new one when $\mathbf{u}[i]$ ($\mathbf{w}[i]$) changes. Line 3 will only be executed if $\text{child}_i(\mathbf{u}[i]) > \text{child}_i(\mathbf{w}[i])$, which will only happen if the tree no longer contains any ith child as a terminal. Note that Lemmas 2.8 and 3.2 imply that if, for some i and T_m, no nonterminal has an ith child in T_m, then no nonterminal has an ith child in T_{m+1}.

Construction of the first tree. Tree $T_{m_{\min}}$ has a simple structure. Its nonterminals are the nodes $\langle 1, 2, \ldots, m_{\min}\rangle$. Its terminals are the n shallowest children of nodes $\langle 1, 2, \ldots, m_{\min}\rangle$.

To construct $T_{m_{\min}}$, we assume that $n > r$; otherwise, $T_{m_{\min}}$ is simply the root and its first n children. For $1 \le m < m_{\min}$, define T_m to be the tree with nonterminals $\{1, \ldots, m\}$ and *all* of the $(r-1)m+1$ children of $\{1, \ldots, m\}$ as terminals. The proof of Lemma 2.7 generalizes easily to these trees; node $m+1$ is the minimum terminal of T_m.

```
CREATE-$T_{m_{\min}}(T)$
    —Create $T_1$—
 1.   $m_{\min} = \lceil \frac{n-1}{r-1} \rceil$; $\mathbf{D}[1] \leftarrow 0$; $\mathbf{C} = \sum_{i=1}^{\min\{r,n\}} c_i$;
 2.   CREATE low-queue; CREATE high-queue;
 3.   FOR $i = 1$ to $\min\{r, n\}$ DO
 4.        $\mathbf{u}[i] \leftarrow \mathbf{w}[i] \leftarrow 1$; UPDATE-QS$(T, i)$;

    —Create $\langle T_2, T_3, \ldots, T_{m_{\min}-1}\rangle$—
 5.   FOR $\mathbf{m} = 2$ to $(m_{\min} - 1)$ DO
 6.        Let $\text{child}_i(\mathbf{u}[i])$ be the minimum terminal in low-queue.
 7.        $\mathbf{D}[\mathbf{m}] \leftarrow \mathbf{D}[\mathbf{u}[i]] + c_i$; $\mathbf{u}[i] \leftarrow \mathbf{u}[i] + 1$; UPDATE-QS$(T, i)$;
 8.        FOR $j = 1$ to $r$ DO
 9.             $\mathbf{w}[j] \leftarrow \mathbf{m}$; UPDATE-QS$(T, j)$;
10.        $\mathbf{C} \leftarrow \mathbf{C} - \mathbf{D}[\mathbf{m}] + \sum_{j=1}^{r} (\mathbf{D}[\mathbf{m}] + c_j)$;

    —Create $T_{m_{\min}}$—
11.   $\mathbf{m} = m_{\min}$; $\Delta = n - (r-1)(m_{\min} - 1)$;
12.   Let $\text{child}_i(\mathbf{u}[i])$ be the minimum terminal in low-queue.
13.   $\mathbf{D}[\mathbf{m}] \leftarrow \mathbf{D}[\mathbf{u}[i]] + c_i$; $\mathbf{u}[i] \leftarrow \mathbf{u}[i] + 1$; UPDATE-QS$(T, i)$
14.   FOR $j = 1$ to $\Delta$ DO
15.        $\mathbf{w}[j] \leftarrow \mathbf{m}$; UPDATE-QS$(T, j)$;
16.   $\mathbf{C} \leftarrow \mathbf{C} - \mathbf{D}[\mathbf{m}] + \sum_{j=1}^{\Delta} (\mathbf{D}[\mathbf{m}] + c_j)$;
17.   $\mathbf{mDeg} = \Delta$;
18.   LEVEL$(T)$;
```

FIG. 7. *Operation* CREATE-$T_{m_{\min}}$.

The tree T_1 is easy to construct. It is the tree with 1 root and r children. Inductively construct the tree T_m from the tree T_{m-1}, $m < m_{\min}$, as follows: find the minimum terminal in T_m by taking the minimum terminal out of **low-queue**. Label this node m, make it a nonterminal, and add all of its children to T_m as terminals. The details are shown in Figure 7.

Finally, construct $T_{m_{\min}}$ from $T_{m_{\min}-1}$ by making the lowest terminal of $T_{m_{\min}-1}$ into node $m_{\min}$. Add the $n - (r-1)(m_{\min} - 1)$ minimum children of node $m_{\min}$ as terminals, bringing the total number of terminals in the current tree to n. Level the resulting tree.

Since only $O(n/r)$ trees are constructed while computing $T_{m_{\min}}$ and each tree can be constructed from the previous tree in $O(r \log r)$ time, the time required to compute $T_{m_{\min}}$ is $O(n \log r)$. (If desired, the time for each tree T_m with $m < m_{\min}$ can be reduced to $O(\log r)$ because maximum terminals are not replaced in constructing such a tree.)

Construction of the remaining trees. The algorithm constructs the sequence of trees

$$\langle T_{m_{\min}}, T_{m_{\min}+1}, T_{m_{\min}+2}, \ldots, T_{m_{\max}} \rangle,$$

as described previously. Tree T_m is found by SPROUTing and then LEVELing its predecessor T_{m-1}. The cost is $O(d_m \log r)$ time, where d_m is the number of children of the new nonterminal m in T_m. By Lemma 3.4, this part of the algorithm runs in $O\left(\left(\sum_m d_m\right) \log r\right) = O(n \log^2 r)$ time.

Acknowledgments. The authors would like to thank Dr. Jacob Ecco for introducing us to the Morse code puzzle, which sparked this investigation. They would also like to thank Siu Ngan Choi and Rudolf Fleischer (who made Observation 3.6—the unimodality of the tree costs) for their careful reading of an earlier manuscript and subsequent comments.

REFERENCES

[1] D. ALTENKAMP AND K. MELHORN, *Codes: Unequal probabilies, unequal letter costs*, J. Assoc. Comput. Mach., 27 (1980), pp. 412–427.

[2] N. COT, *Complexity of the variable-length encoding problem*, in Proc. 6th Southeast Conference on Combinatorics, Graph Theory, and Computing, Congressus Numerantium XIV, Utilitas Mathematica Publishing, Winnepeg, MB, Canada, 1975, pp. 211–244.

[3] D. A. HUFFMAN, *A method for the construction of minimum redundancy codes*, Proc. IRE, 40 (1952), pp. 1098–1101.

[4] S. KAPOOR AND E. M. REINGOLD, *Optimum lopsided binary trees*, J. Assoc. Comput. Mach., 36 (1989), pp. 573–590.

[5] R. KARP, *Minimum-redundancy coding for the discrete noiseless channel*, IRE Trans. Inform. Theory, IT-7 (1961), pp. 27–39.

[6] D. E. KNUTH, *The Art of Computer Programming, Volume* III: *Sorting and Searching*, Addison–Wesley, Reading, MA, 1973.

[7] Y. PERL, M. R. GAREY, AND S. EVEN, *Efficient generation of optimal prefix code: Equiprobable words using unequal cost letters*, J. Assoc. Comput. Mach., 22 (1975), pp. 202–214.

[8] B. VARN, *Optimal variable length codes (arbitrary symbol cost and equal code word probability)*, Inform. Control, 19 (1971), pp. 289–301.

SIAM J. COMPUT.
Vol. 25, No. 6, pp. 1293–1304, December 1996

009

ON UNAPPROXIMABLE VERSIONS OF *NP*-COMPLETE PROBLEMS*

DAVID ZUCKERMAN†

Abstract. We prove that all of Karp's 21 original *NP*-complete problems have a version that is hard to approximate. These versions are obtained from the original problems by adding essentially the same simple constraint. We further show that these problems are absurdly hard to approximate. In fact, no polynomial-time algorithm can even approximate $\log^{(k)}$ of the magnitude of these problems to within any constant factor, where $\log^{(k)}$ denotes the logarithm iterated k times, unless NP is recognized by slightly superpolynomial randomized machines. We use the same technique to improve the constant ϵ such that MAX CLIQUE is hard to approximate to within a factor of n^ϵ. Finally, we show that it is even harder to approximate two counting problems: counting the number of satisfying assignments to a monotone 2SAT formula and computing the permanent of -1, 0, 1 matrices.

Key words. *NP*-complete, unapproximable, randomized reduction, clique, counting problems, permanent, 2SAT

AMS subject classifications. 68Q15, 68Q25, 68Q99

1. Introduction.

1.1. Previous work. The theory of *NP*-completeness was developed in order to explain why certain computational problems appeared intractable [10, 14, 12]. Yet certain optimization problems, such as MAX KNAPSACK, while being *NP*-complete to compute exactly, can be approximated very accurately. It is therefore vital to ascertain how difficult various optimization problems are to approximate.

One problem that eluded attempts at accurate approximation is MAX CLIQUE. This is the problem of finding $\omega(G)$, the size of a largest clique in the graph G. There was no explanation for this until Feige et al. [11] showed that for all $\epsilon > 0$, no polynomial-time algorithm can approximate $\omega(G)$ to within a factor of $2^{(\log n)^{1-\epsilon}}$ unless $N\tilde{P} = \tilde{P}$, where $\tilde{P}$ denotes quasi-polynomial time, or $TIME(2^{\text{polylog}})$. This was based on the proof that MIP = NEXP [5]. Recently, there have been several improvements, culminating in the result that approximating $\omega(G)$ to within a factor of $n^{1/4-o(1)}$ is *NP*-complete [4, 3, 7].

1.2. A new role for old reductions. It is natural and important to identify other *NP*-complete problems that are hard to approximate. In the original theory of *NP*-completeness, polynomial-time reductions were used. Yet these reductions might not preserve the quality of an approximation well, so researchers focused on reductions that preserved the quality of approximation very closely [18, 17]. Using such reductions, Panconesi and Ranjan [17] defined a class RMAX(2) of optimization problems, of which MAX CLIQUE is one natural complete problem. The intractability of approximating MAX CLIQUE implies that the other RMAX(2)-complete problems are intractable to approximate. Recently, Lund and Yannakakis

* Received by the editors April 21, 1994; accepted for publication (in revised form) March 14, 1995. A preliminary version of this paper, entitled "*NP*-complete problems have a version that's hard to approximate," appeared in *Proc. 8th IEEE Conference on Structure in Complexity Theory*, IEEE Press, Piscataway, NJ, 1993, pp. 305–312. Most of this research was done while the author was supported by an NSF Postdoctoral Fellowship at MIT. At UT Austin, the author was partially supported by NSF NYI grant CCR-9457799.

† Department of Computer Sciences, University of Texas at Austin, Austin, TX 78712 (diz@cs.utexas.edu).

[16] used an approximation-preserving reduction to show the intractability of approximating CHROMATIC NUMBER.

Here we show that the reductions can have a much more general form and still yield unapproximability results. This is essentially because the factors are huge, namely n^ϵ, so polynomial blow-ups will only change this to $n^{\epsilon'}$. We show that Karp's original reductions can be modified to have this general form, and hence the original 21 NP-complete problems presented in [14] all have a version that is hard to approximate. This gives evidence that all NP-complete problems have a version that is hard to approximate.

1.3. The small jump to unapproximability. What does it mean that an NP-complete problem has a version that's hard to approximate? Indeed, an NP-complete problem is a language-recognition problem and may not even have a corresponding optimization problem; moreover, many corresponding optimization problems are easy to approximate. Intuitively, however, it seems reasonable that by adding sufficiently many constraints to an optimization problem, it becomes "harder," i.e., hard to approximate. Quite surprisingly, we show that by adding one simple constraint that is essentially the same for every NP-complete problem, all of Karp's original NP-complete problems become unapproximable.

What is this constraint? Usually, we can only form maximization problems when our NP language L is of the form $(x, k) \in L \iff (\exists y, f(y) \geq k) p(x, y)$ for some polynomial-time predicate p and function f. The corresponding optimization problem is then taken to be $\max_{y:p(x,y)} f(y)$. Of course, if L is of the same form except with $f(y) \leq k$, then we end up with the minimization problem $\min_{y:p(x,y)} f(y)$. For example, for the NP-complete language VERTEX COVER, x is a graph and y is a set of vertices; p is the predicate that y forms a vertex cover in x and $f(y)$ is the size of y. Thus the language is "Does there exist a vertex cover of size k?" while the optimization problem is to find the size of a minimum vertex cover.

Although we do not prove a general theorem for all NP-complete languages, the constraint we add makes sense for any NP language. This is because we use the basic representation of an NP language L as $x \in L \iff (\exists y \in \{0,1\}^m) p(x, y)$, where m is polynomial in the length of x. Note that for languages that can be expressed as in the previous paragraph, the x here is what was previously the ordered pair (x, k) and the p here is what was previously $p(x, y) \wedge (f(y) \geq k)$ (or, for minimization problems, $f(y) \leq k$). The constraint we add is as follows. Using the natural correspondence between $\{0,1\}^m$ and subsets of $\{1, \ldots, m\}$, we view y as a subset of $\{1, \ldots, m\}$. We have as an additional input a subset $S \subseteq \{1, \ldots, m\}$, and the output should be $\max_{y \in \{0,1\}^m : p(x,y)} |S \cap y|$. We also insist that there be a y such that $p(x, y)$; otherwise, by taking $S = \{1, \ldots, m\}$, deciding whether the maximum is 0 or not is equivalent to deciding whether there exists a y such that $p(x, y)$. We even give such a y for free as an additional input.

Thus, from the easily approximable minimization problem VERTEX COVER, we obtain the unapproximable constrained maximization version: given a graph $G = (V, E)$, $S \subseteq V$, a positive integer k, and a vertex cover of size at most k, find the maximum of $|S \cap C|$ over vertex covers C of size at most k. This can be interpreted as follows: the set S represents the important vertices; we can only afford a vertex cover of size at most k but wish to use as many important vertices as possible.

For HAMILTONIAN CIRCUIT, this constrained version becomes the following: given a graph $G = (V, E)$, a Hamiltonian circuit in G, and $S \subseteq E$, find the maximum of $|S \cap C|$ over Hamiltonian circuits C. This has a natural interpretation: a salesman has to visit all cities, and certain trips between cities are scenic or free, so he wants to maximize the number of such trips.

1.4. Hyperunapproximability results. Not only are these versions hard to approximate to within a factor of n^ϵ, but it is also hard to have any idea about the order of magnitude of the solutions to these optimization problems. More specifically, we show that any iterated logarithm of any of the above versions is hard to approximate within a constant factor unless NP is recognized by slightly superpolynomial randomized machines. (Slightly superpolynomial will be made precise in the next paragraph.) The proof also does not rely on the fact that the iterated logarithm may become 0 (or negative); we can assume the iterated logarithm is at least 1. This result extends the result in [22] that the logarithm of $\omega(G)$ is hard to approximate to within any constant factor unless $\tilde{NP} = \tilde{P}$.[1]

In order to state our results precisely, we define

$$\log^{(k)} n = \underbrace{\log\log\cdots\log}_{k} n,$$

$$p_{e,k}(n) = 2^{2^{\cdot^{\cdot^{2^{e\log^{(k)} n}}}}} \Bigg\} k \text{ 2's},$$

$$P_k = \bigcup_{e>0} TIME(p_{e,k}(n)),$$

with analagous definitions for NP_k, RP_k, co-RP_k, and $ZPP_k = RP_k \cap$ co-RP_k. Note that $P_1 = P$ is polynomial time, $P_2 = \tilde{P}$ is quasi-polynomial time, and P_k for $k > 2$ are other measures of slightly superpolynomial time. Also, let FC denote the functional version of the complexity class C. In particular, $FZPP_k$ corresponds to functions computable by zero-error (Las Vegas) randomized algorithms that run in the appropriate expected time. For a function f, the notation f(MAX CLIQUE) denotes the problem of finding $f(\omega(G))$, and similarly for other optimization problems. We show that if $NP_k \neq ZPP_k$, then no function in $FZPP_k$ approximates $\log^{(k)}$ of any of the above versions of NP-complete problems (e.g., MAX CLIQUE) to within any constant factor.

These techniques can also be used to improve the constant ϵ such that MAX CLIQUE cannot be approximated to within a factor n^ϵ. Suppose c answer bits are required by a PCP protocol to achieve error 1/2. We show that if $NP \neq ZPP$, then for all $\epsilon < 1/(c+1)$, there is no Las Vegas algorithm running in expected polynomial time which approximates MAX CLIQUE to within a factor n^ϵ. Recently, much effort has been devoted towards improving the constant (see, e.g., [6, 7]), and they all use this lemma or an extension of it.

We point out that similar results may be obtained by using the randomized graph product method of Berman and Schnitger [8]. However, such results would be under the stronger assumption that $NP_k \neq BPP_k$. The reason for this is that we look at the proof-theoretic construction of the graphs in question, while Berman and Schnitger use a straight reduction. We therefore need only deal with the error in the "easy" direction, while Berman and Schnitger need to worry about the error in both directions.

[1] This does not entirely improve upon [22]; here we show that $\log\omega(G)$ is hard to approximate unless $NP = ZPP$, a condition which, as far as we know, does not imply $\tilde{NP} = \tilde{P}$.

This difference also manifests itself in the derandomization: more work is needed to derandomize the randomized graph product construction [2] than the basic tool used to derandomize the proof-theoretic construction [1].

1.5. Implications for counting problems. We further show that under the same assumption that $NP_k \neq ZPP_k$, $\log^{(k+1)}$ of the number of satisfying assignments to a monotone 2SAT formula is hard to approximate to within any constant factor. That this is hard to approximate may seem surprising because finding a satisfying assignment is trivial. In the case of a DNF formula, where finding a satisfying assignment is also easy, approximating the number of satisfying assignments is in randomized polynomial time [15].

As a corollary, we use Valiant's reduction [21] to observe that approximating $\log^{(k+1)}$ of the permanent of a matrix with entries in $\{-1, 0, 1\}$ is hard under the same assumption as above. We can assume the matrix has positive permanent because, conceivably, the problem of deciding if the permanent is 0 is NP-hard, which would make the corollary uninteresting. This result should be contrasted with the subexponential algorithm to approximate the permanent of 0,1-matrices [13].

2. The iterated log of max clique is hard to approximate.

In this section, we show that it is hard to approximate any iterated logarithm of the size of the maximum clique. We first present the following definition.

DEFINITION 2.1. Approximating $g(x)$ to within a factor $a(n)$ is in $FZTIME(t(n))$ *if there is a zero-error (Las Vegas) randomized algorithm which, on input* x, *runs in expected time* $t(|x|)$ *and outputs* $\tilde{g}$ *such that* $g(x)$ *is in the half-open interval* $I = [\tilde{g}, a(|x|)\tilde{g})$. *Here* $|x|$ *denotes the length of* x.

Thus the algorithm can distinguish between x and y, $|x| = |y| = n$, if $g(x) \geq a(n)g(y)$ or $g(y) \geq a(n)g(x)$.[2]

Our proofs closely follow the proofs of [11, 4, 3], building on the work of [5]. First, here are some definitions from [4].

A verifier is a probabilistic polynomial-time probabilistic Turing machine M given access to the input x, random bits y, and a proof Π. The verifier's goal is to decide whether Π is a valid proof that x is in some language L. We define the predicate $M^{\Pi}(x, y)$ to be true iff M accepts x given the proof Π and random bits y.

DEFINITION 2.2. *A verifier is* $(r(n), c(n))$*-restricted if on an input of size* n *it uses at most* $r(n)$ *random bits and queries at most* $c(n)$ *bits of the proof.*

DEFINITION 2.3. *A language* L *is in the complexity class* $PCP(r(n), c(n))$ *iff there is an* $(r(n), c(n))$*-restricted verifier such that*

$$\begin{aligned} x \in L &\Rightarrow (\exists \Pi) Pr_y[M^{\Pi}(x, y)] = 1, \\ x \notin L &\Rightarrow (\forall \Pi) Pr_y[M^{\Pi}(x, y)] < 1/2. \end{aligned}$$

Arora et al. [3] give the following improvement of [5].

THEOREM 2.4 (see [3]). $NP = PCP(O(\log n), O(1))$.

Using this, they follow [11] and construct a graph G_x which has a large clique iff $x \in L$. In order to do this, they define transcripts and a notion of consistency among

[2] Of course, our definition is also equivalent to the algorithm always outputting a number $\tilde{g}$ such that $g(x)/b(|x|) \leq \tilde{g} < b(|x|)g(x)$, where $b(n) = \sqrt{a(n)}$. One might think it is natural to define this as approximating to within a factor $b(n)$; however, we want the property that the algorithm can distinguish between g values that differ by an $a(n)$ factor. Otherwise, we will get $b(n)^2$ terms instead of $a(n)$ terms. Our definition is also like the ones used in [11, 3].

them. A transcript is basically a set of queries to locations of the proof and the bits that are found in these locations. Two transcripts are consistent if there is one proof that can correspond to both transcripts.

DEFINITION 2.5 (see [11]). *We say that $\langle y, q_1, a_1, \ldots, q_c, a_c\rangle$ is an (r,c)-*transcript *for verifier M on x if $|y| = r$, and on input x and random string y, for every i, M queries bit location q_i after receiving the answers a_j to queries q_j for $j < i$. The transcript is* accepting *if on input x, random string y, and history of communication (questions and answers) $\langle q_1, a_1, \ldots, q_c, a_c\rangle$, M accepts x.*

DEFINITION 2.6 (see [11]). *We say that two transcripts $\langle y, q_1, a_1, \ldots, q_c, a_c\rangle$ and $\langle \hat{y}, \hat{q_1}, \hat{a_1}, \ldots, \hat{q_c}, \hat{a_c}\rangle$ are consistent if for every i, $q_i = \hat{q}_i$ implies $a_i = \hat{a}_i$.*

To decide whether x is in some NP language L, we construct a graph G_x based on the $(r(n) = O(\log n), c(n) = O(1))$-restricted verifier M for L. The vertices of G_x are all accepting $(r(n), c(n))$-transcripts of M on x, and two nodes are connected iff the corresponding transcripts are consistent. Thus G_x has at most $2^{r(n)+c(n)}$ vertices. The following result is also not hard to see.

LEMMA 2.7 (see [11]). $\omega(G_x) = \max_\Pi \Pr_y[M^\Pi(x,y)] \cdot 2^{r(n)}$.

In other words, $\omega(G_x)$ is the maximum over all proofs Π of the number of random strings on which M accepts x. Thus if $x \in L$, then $\omega(G_x) = 2^{r(n)}$, and if $x \notin L$, then $\omega(G_x) < 2^{r(n)}/2$.

In order to get a wider separation in the clique sizes, Feige et al. constructed the graph G'_x corresponding to a protocol M'. M' runs $\log^{O(1)} n$ independent iterations of M on x. This reduces the error probability if $x \notin L$ and therefore produces a wider separation in the clique sizes.

Yet once we fix a proof Π, M^Π basically corresponds to a co-RP machine: always accepting when $x \in L$ and usually rejecting if $x \notin L$. Thus it is natural to use pseudorandom strings that efficiently amplify the success probability of an RP (or co-RP) algorithm. Indeed, this was the idea used in [22] to show that approximating $\log \omega$ is hard. Arora et al. [3] later used this idea to achieve their result as well.

But since we will cycle through all possibilities of the random seeds, the pseudorandom strings do not have to be constructible in the usual sense. In fact, the best amplification schemes are given by random graphs, which are so-called "dispersers" with high probability. Thus our plan will be to use a random amplification scheme.

DEFINITION 2.8. *An (R, r, d)-amplification scheme is a bipartite graph $H = (\{0,1\}^R \cup \{0,1\}^r, E)$, where $\{0,1\}^R$ and $\{0,1\}^r$ are independent sets and the degree of every node in $\{0,1\}^R$ is d.*

An (R, r, d)-amplification scheme defines a pseudorandom generator that takes as input an R-bit string z and outputs the d r-bit neighbors of z. A good amplification scheme has been called a disperser [9].

DEFINITION 2.9. *An (m, n, d, a, b)-disperser is a bipartite graph with m nodes on the left side, each with degree d, and n nodes on the right side such that every subset of a nodes on the left side has at least b neighbors on the right.*

We pick an (R, r, d)-amplification scheme uniformly at random by choosing independently, for each $u \in \{0,1\}^R$, d uniformly random elements from $\{0,1\}^r$ as the neighbors of u. Santha [19] and Sipser [20] have shown that a random amplification scheme is a disperser. In order to get the optimal ϵ in the n^ϵ results, we use an extremely minor modification of their arguments.

LEMMA 2.10. *The probability that a uniformly random $(R, r, R+2)$-amplification scheme is a $(2^R, 2^r, R+2, 2^r, 2^{r-1})$-disperser is greater than $1 - 2^{-2^r}$.*

Proof. We basically follow [20]. For $S \subseteq \{0,1\}^R$, $T \subseteq \{0,1\}^r$, let $A_{S,T}$ be the event that all neighbors of S are in T. Then the probability that the amplification scheme is not the desired disperser equals

$$\Pr\left[\bigcup_{\substack{|S|=2^r \\ |T|=2^{r-1}-1}} A_{S,T}\right] \leq \sum_{\substack{|S|=2^r \\ |T|=2^{r-1}-1}} \Pr[A_{S,T}] = \binom{2^R}{2^r}\binom{2^r}{2^{r-1}-1}\left(\frac{2^{r-1}-1}{2^r}\right)^{(R+2)2^r}$$

$$< 2^{R2^r} \cdot 2^{2^r} \cdot 2^{-(R+2)2^r} = 2^{-2^r}. \qquad \square$$

The unapproximability of the iterated log will follow from the following lemma.

LEMMA 2.11. *Let $L \in PCP(r(n) = O(\log n), c)$. Let $R = R(n)$ be a function of n, and let $N = N(n) = 2^{R+(R+2)c}$. If there is a Las Vegas algorithm running in expected time $t(N)$ which can correctly determine whether a graph with at most N nodes has a clique of size at least 2^R or at most 2^r, then $L \in$ co-$RTIME(t(N(n)) + p(N(n)))$ for some polynomial p.*

Proof. We describe a randomized algorithm A to recognize L. Let x be the input. Let M be the $(r = r(n), c)$-restricted verifier accepting L. A picks a uniformly random $(R, r, R + 2)$-amplification scheme H. Define the verifier V as the machine which picks a uniformly random R-bit string, determines its neighbors $y_1, \ldots, y_{R+2}$ in H, and simulates M on x with the random strings $y_1, \ldots, y_{R+2}$. V accepts iff all $R + 2$ runs of M accept. Note that V uses R random bits and $dc = (R + 2)c$ answer bits. A constructs G'_x corresponding to V. Then G'_x has at most $N = 2^{R+(R+2)c}$ vertices. Observe that if $x \in L$, then $\omega = \omega(G'_x) = 2^R$. Now consider if $x \notin L$. Let Π be an arbitrary proof. With overwhelming probability (in particular, at least 5/6), H is a $(2^R, 2^r, d, 2^r, 2^{r-1})$-disperser. Since less than 2^{r-1} r-bit strings cause M to accept, by the disperser property of H, at most 2^r R-bit strings cause V to accept. Thus $\omega \leq 2^r$.

Let B be a Las Vegas algorithm running in expected time $t(N)$ which correctly determines whether $\omega \geq 2^R$ or $\omega \leq 2^r$. A simulates B for $3t(N)$ steps (in which time B fails to halt with probability at most 1/3). If B doesn't halt or determines that $\omega \geq 2^R$, then A accepts. Otherwise, A rejects. Thus A always accepts if $x \in L$ and with probability $\geq 5/6 - 1/3$ rejects if $x \notin L$. A runs in time $O(t(N) + p(N))$, where p is a polynomial depending on the running time of M. $\square$

We also make use of a complexity-theoretic lemma.

LEMMA 2.12. *For any positive integer k, $NP \subseteq$ co-RP_k iff $NP_k = ZPP_k$.*

Proof. One direction is easy: if $NP_k = ZPP_k$, then $NP \subseteq NP_k = ZPP_k \subseteq$ co-RP_k. Now assume $NP \subseteq$ co-RP_k. Since $p_{d,k}(p_{e,k}(n)) \leq p_{de,k}(n)$, this implies that $NP_k \subseteq$ co-RP_k. Taking complements, we get co-$NP_k \subseteq RP_k$. But then $NP_k \subseteq$ co-$RP_k \subseteq$ co-$NP_k \subseteq RP_k \subseteq NP_k$. Thus all containments are equalities and $NP_k = RP_k =$ co-RP_k; hence $NP_k = ZPP_k$. $\square$

We can now show the following.

THEOREM 2.13. *If for any constant a approximating $\log^{(k)} \omega$ to within a factor a is in $FZPP_k$, then $NP_k = ZPP_k$.*

Proof. Let $L \in PCP(r(n) = O(\log n), c)$ be NP-complete. Set $R = p_{a,k-1}(r)$ and apply Lemma 2.11. Suppose there is an algorithm A approximating $\log^{(k)} \omega$ to within a factor a. Since $\log^{(k)} 2^R = a \cdot \log^{(k)} 2^r$, A can determine whether $\omega \geq 2^R$ or $\omega \leq 2^r$ in a graph on $N = 2^{R+(R+2)c}$ vertices. For n large enough so that $R \geq 2c$, $N \leq 2^{(c+2)R} = 2^{(c+2)p_{a,k-1}(r)} \leq 2^{(c+2)p_{a,k-1}(\log n)} \leq p_{a(c+2),k}(n)$. Thus, for some constant e, A runs in time $p_{e,k}(N) \leq p_{ea(c+2),k}(n)$ on inputs of length n. Lemma

2.11 now implies the theorem, except that the conclusion is $NP \subseteq$ co-RP_k instead of $NP_k = ZPP_k$. Lemma 2.12 shows that these conclusions are equivalent. □

Similarly, we improve the constant ϵ in the n^ϵ of the MAX CLIQUE unapproximability results of [3].

THEOREM 2.14. *Let c be a constant such that some NP-complete language is in* $PCP(O(\log n), c)$ *(which exists by Theorem* 2.4*). Then for any constant* $\epsilon < 1/(c+1)$*, there is no Las Vegas algorithm running in expected polynomial time that approximates MAX CLIQUE to within a factor* n^ϵ *unless* $NP = ZPP$.

Proof. Choose k large enough so that $(k-1)/(k+(k+2)c) \geq \epsilon$, and let $R = kr$. By Lemma 2.11, ω cannot be approximated to within a factor of 2^{R-r} in a graph with $N = 2^{R+(R+2)c}$ vertices unless $NP \subseteq$ co-RP. By our choice of R, this factor is at least N^ϵ. Moreover, by Lemma 2.12, $NP \subseteq$ co-RP is equivalent to $NP = ZPP$. □

3. Unapproximable versions of NP-complete problems.

We now modify Karp's list of 21 NP-complete problems to obtain versions that are hard to approximate. Problems 4 and 11 had previously been shown to be as difficult to approximate as MAX SAT [17].

THEOREM 3.1. *For each of the following maximization problems A, there exists a constant* $\epsilon > 0$ *such that A cannot be approximated to within a factor* n^ϵ *in polynomial time unless* $P = NP$*. For any positive constant c, any positive integral k, and any of the following maximization problems A, approximating* $\log^{(k)}(A)$ *to within a factor of c is not in* $FZPP_k$ *unless* $NP_k = ZPP_k$.

1. **CONSTRAINED MAX SAT**
   See MAX 2ANLSAT.
2. **MAX 0–1 INTEGER PROGRAMMING**
   Input: integer matrix C and integer vector d
   Output: the maximum, over all 0–1 vectors x such that $Cx \geq d$, of the number of 1's in x.
3. **MAX CLIQUE**
   Input: undirected graph G
   Output: the maximum number of vertices in a clique.
4. **MAX SET PACKING**
   Input: family of finite sets $\{S_j\}$, $j \in \{1, \dots, n\}$
   Output: the maximum, over all $I \subseteq \{1, \dots, n\}$ such that the $S_i, i \in I$ are disjoint, of $|I|$.
5. **CONSTRAINED MAX VERTEX COVER**
   Input: undirected graph $G = (V, E)$, $S \subseteq V$, and a vertex cover of size k
   Output: the maximum, over vertex covers C of size k, of $|C \cap S|$.
6. **CONSTRAINED MAX SET COVERING**
   Input: family of finite sets $\{S_i\}$, $i = 1, \dots, n$, $T \subseteq \{1, \dots, n\}$, and a subcover of size k (i.e., $J \subseteq \{1, \dots, n\}$, $|J| = k$, such that $\cup_{j \in J} S_j = \cup_{i=1}^n S_i$)
   Output: the maximum, over subcovers C of size k, of $|C \cap T|$.
7. **CONSTRAINED MAX FEEDBACK NODE SET**
   Input: digraph $G = (V, E)$, $S \subseteq V$, and a feedback node set of size k (i.e., a subset $R \subseteq V$ of size k that contains a vertex of every directed cycle)
   Output: the maximum, over feedback node sets C of size k, of $|C \cap S|$.

8. **CONSTRAINED MAX FEEDBACK ARC SET**
Input: digraph $G = (V, E)$, $S \subseteq E$, and a feedback arc set of size k (i.e., a subset $R \subseteq E$ of size k that contains an edge of every directed cycle)
Output: the maximum, over feedback arc sets C of size k, of $|C \cap S|$.

9. **CONSTRAINED MAX DIRECTED HAMILTONIAN CIRCUIT**
Input: digraph $G = (V, E)$, $S \subseteq E$, and a Hamiltonian circuit in G
Output: the maximum, over Hamiltonian circuits $C \subseteq E$, of $|C \cap S|$.

10. **CONSTRAINED MAX HAMILTONIAN CIRCUIT**
Input: undirected graph $G = (V, E)$, $S \subseteq E$, and a Hamiltonian circuit in G
Output: the maximum, over Hamiltonian circuits $C \subseteq E$, of $|C \cap S|$.

11. **MAX 2ANLSAT**
Input: 2CNF formula F with all variables negated
Output: the maximum, over all satisfying assignment x, of the number of variables set to "true" in x.

12. **CONSTRAINED MAX CHROMATIC NUMBER**
Input: graph $G = (V, E)$, $v \in V$, $S \subseteq V$, and a k-coloring of G
Output: the maximum, over all k-colorings $\mathcal{C}$ of G, of $|C \cap S|$, where C is the set of vertices in the same color class as v in $\mathcal{C}$.

13. **CONSTRAINED MAX CLIQUE COVER**
Input: graph $G = (V, E)$, $v_0 \in V$, $S \subseteq V$, and a clique cover of G of size at most k, i.e., a representation of G as the union of at most k cliques
Output: the maximum, over all clique covers $\mathcal{C}$ of G of size at most k, of $|C \cap S|$, where C is the clique containing v in $\mathcal{C}$.

14. **CONSTRAINED MAX EXACT COVER**
Input: family of finite sets $\{S_i\}$, $i = 1, \ldots, n$, $T \subseteq \{1, \ldots, n\}$, and an exact cover (i.e., $J \subseteq \{1, \ldots, n\}$ such that the S_j, $j \in J$ are disjoint, and $\cup_{j \in J} S_j = \cup_{i=1}^{n} S_i$)
Output: the maximum, over exact covers C, of $|C \cap T|$.

15. **CONSTRAINED MAX HITTING SET**
Input: family of subsets $\{S_i\}$ of $\{i = 1, \ldots, n\}$, $T \subseteq \{1, \ldots, n\}$, and a hitting set (i.e., $W \subseteq \{1, \ldots, n\}$ such that for all i, $|W \cap \{1, \ldots, n\}| = 1$)
Output: the maximum, over hitting sets C, of $|C \cap T|$.

16. **CONSTRAINED MAX STEINER TREE**
Input: undirected graph $G = (V, E)$, $S \subseteq E$, and a Steiner tree of weight at most k with respect to $R \subseteq V$ and weighting function $w : E \to Z$ (i.e., a subtree of weight at most k containing the set of nodes in R)
Output: the maximum, over Steiner trees $T \subseteq E$ of weight at most k with respect to R and w, of $|T \cap S|$.

17. **CONSTRAINED MAX THREE-DIMENSIONAL MATCHING**
Input: hypergraph $H = (V, F)$, $F \subseteq V \times V \times V$, $S \subseteq F$, and a three-dimensional matching (i.e., $M \subseteq F$, $|M| = |V|$, and no two elements of M agree in any coordinate)
Output: the maximum, over three-dimensional matchings N, of $|N \cap S|$.

18. **CONSTRAINED MAX KNAPSACK**
Input: $(a_1, a_2, \ldots, a_n) \in Z^n$, $T \subseteq \{1, \ldots, n\}$, and a knapsack of size b (i.e., an $S \subseteq \{1, \ldots, n\}$, $\sum_{j \in S} a_j = b$)
Output: the maximum, over knapsacks C of size b, of $|C \cap T|$.

19. **CONSTRAINED MAX JOB SEQUENCING**
Input: "execution time vector" $(T_1, \ldots, T_n) \in Z^n$
"deadline vector" $(D_1, \ldots, D_n) \in Z^n$
"penalty vector" $(P_1, \ldots, P_n) \in Z^n$
$S \subseteq \{1, \ldots, n\}$ and a schedule (permutation) π with penalty at most k, i.e.,

$$\sum_j [\text{if } T_{\pi(1)} + \cdots + T_{\pi(j)} > D_{\pi(j)} \text{ then } P_{\pi(j)} \text{ else } 0] \leq k$$

Output: the maximum, over schedules with penalties of at most k, of the number of jobs in S completed by the deadline.

20. **CONSTRAINED MAX PARTITION**
Input: $(a_1, a_2, \ldots, a_n) \in Z^n$, $k \in \{1, \ldots, n\}$, $T \subseteq \{1, \ldots, n\}$, and an equal partition (i.e., an $S \subseteq \{1, \ldots, n\}$, $\sum_{j \in S} a_j = \sum_{j \notin S} a_j$)
Output: the maximum, over equal partitions C with $k \in C$, of $|C \cap T|$.

21. **CONSTRAINED MAX CUT**
Input: undirected graph $G = (V, E)$, $v \in V$, $T \subseteq V$, and a cut of weight at least W with respect to the weighting function $w : V \rightarrow Z$ (i.e., a set $S \subseteq V$ such that

$$\sum_{\{u,v\} \in E, u \in S, v \notin S} w(\{u, v\}) \geq W)$$

Output: the maximum, over cuts C of weight at least W with $v \in C$, of $|C \cap T|$.

Note that the above languages are all of the following similar form. Let p be a polynomial-time predicate corresponding to an NP language L so that $x \in L$ iff $(\exists y \in \{0,1\}^m) p(x, y)$, where m is polynomial in n. Let $S \subseteq \{1, \ldots, m\}$, and view y as a subset of $\{1, \ldots, m\}$. Then the maximization problems above correspond to maximizing $|S \cap y|$ over y such that $p(x, y)$, given such a y.

We now consider when reductions between two such maximization problems preserve the difficulty of approximation.

LEMMA 3.2. *Suppose L is a language of the above form, where approximating $\beta = \max\{|S \cap y|\}$, given some y such that $p(x, y)$, to within a factor n^ϵ is hard for some $\epsilon > 0$, and approximating $\log^{(k)} \beta$ to within any constant factor is hard. Let q be a polynomial-time reduction such that $x \in L$ iff $x' = q(x) \in L'$; moreover, given y such that $p(x, y)$ in polynomial time, one can compute y' such that $p'(x', y')$. Suppose that there is an $S' \subseteq \{1, \ldots, m'\}$ such that*

$$(\exists y) p(x, y) \text{ and } |S \cap y| = k \iff (\exists y') p'(x', y') \text{ and } |S' \cap y'| = k.$$

Then approximating $\beta' = \max\{|S' \cap y'|\}$, given some y' such that $p'(x', y')$, to within a factor $n^{\epsilon'}$ is hard for some $\epsilon' > 0$, and approximating $\log^{(k)} \beta'$ to within any constant factor is hard.

Proof. The lemma follows because $\beta' = \beta$ and $|x'| = \text{poly}(|x|)$. □

We can now prove the theorem. We first observe as in [17] that approximating MAX 2ANLSAT is as hard as approximating MAX CLIQUE.

LEMMA 3.3 (see [17]). *For any functions f and g, approximating $f(MAX\ CLIQUE)$ to within a factor $g(n)$ is polynomial-time reducible to approximating $f(MAX\ 2ANLSAT)$ to within a factor $g(n)$.*

Proof. The proof is contained in the proof of Theorem 4.1. □

Proof of Theorem 3.1. We basically use the sequence of reductions given by Karp [14] that the unconstrained versions of the above problems are NP-complete. Lemma 3.3 tells us that the constrained version of 2SAT is hard to approximate. Moreover, for 2SAT, we can easily compute a satisfying assignment if one exists. Next, for most of the problems above, we can look at the reductions in [14] and verify that they satisfy the conditions of Lemma 3.2. There are some reductions, however, for which the reductions in [14] will not work. For example, Karp reduces CLIQUE to VERTEX COVER by taking complements. This would yield a minimization problem. Instead, we use the reduction given in [12] which goes directly from 3SAT, and we can let S be the subset of vertices which Garey and Johnson call u_i.

To show the result for HAMILTONIAN CIRCUIT requires some care. We modify the reduction given in [12] reducing VERTEX COVER to HAMILTONIAN CIRCUIT. We briefly outline their reduction. Say we have an instance of VERTEX COVER: a graph $G = (V, E)$ and an integer k. They construct $G' = (V', E')$ as follows. V' consists of k "selector vertices" $A = \{a_1, \ldots, a_k\}$ plus other vertices corresponding to edges in G. E' is constructed in such a way that G' has a Hamiltonian circuit iff G has a vertex cover of size k. Each a_i has the same adjacency list, and there are no edges between any two a_i.

Our reduction is from MAX INDEPENDENT SET to CONSTRAINED MAX HAMILTONIAN CIRCUIT. Given an instance $G = (V, E)$, $|V| = n$, of MAX INDEPENDENT SET, construct G' using the reduction from VERTEX COVER above with the parameter $k = n$. Form G'' by adding the edges $\{a_i, a_j\}$ for each pair of selector vertices $(a_i, a_j), i < j$. Let S be the edges $\{a_i, a_j\}$ of this clique A. Since there is always a vertex cover of size n in G, there will always be a Hamiltonian circuit C in G' and hence in G''. The construction of [12] ensures that C can be found efficiently. The input to CONSTRAINED MAX HAMILTONIAN CIRCUIT is G'', S, and C.

We show that the output of CONSTRAINED MAX HAMILTONIAN CIRCUIT is α, the size of a maximum independent set in G. That is, we show that there is a Hamiltonian circuit passing through α edges of S and no Hamiltonian circuit passing through $\alpha + 1$ edges of S. We use the fact that the size of a minimum vertex cover is $n - \alpha$. Since there is a vertex cover of size $n - \alpha$ in G, there is a Hamiltonian circuit in G'' which passes through α edges in S. Namely, this is the Hamiltonian circuit in [12] with $a_{n-\alpha}$ replaced by the path $a_{n-\alpha}, a_{n-\alpha+1}, \ldots, a_n$. Note that we can make this replacement since each a_i is connected to the same vertices outside A. Conversely, suppose there is a Hamiltonian circuit in G'' passing through $\alpha + 1$ edges in S. Since each a_i has the same adjacency list outside A, by contracting these edges, we see that there is a Hamiltonian circuit passing through $n - \alpha - 1$ selector vertices in the original construction of [12], and hence there is a vertex cover of size $n - \alpha - 1$, a contradiction. □

4. Two unapproximable counting problems.

In this section, we show how difficult it is to approximate the number of satisfying assignments to a monotone 2CNF formula or, equivalently, a 2CNF formula where all variables are negated. As a corollary, we deduce the hardness of approximating the permanent of a matrix with $\{-1, 0, 1\}$ entries.

THEOREM 4.1. *There exists $\epsilon > 0$ such that if the log of the number of satisfying assignments to a monotone $2CNF$ can be approximated to within a factor of n^ϵ, then $NP = P$. If, for some constant a, approximating $\log^{(k+1)}$ of the number of satisfying assignments to a monotone $2CNF$ to within a factor a is in $FZPP_k$, then $NP_k = ZPP_k$.*

Proof. The proof extends the reduction in [17]. Let $G = (\{1,\dots,n\},E)$ be a graph with maximum clique size $\omega > 1$, and consider the formula $F = \bigwedge_{\{i,j\}\notin E}(\bar{x}_i \vee \bar{x}_j)$. Viewing an assignment x as a subset S_x of $\{1,\dots,n\}$, we see that x satisfies F iff S_x forms a clique in G. Thus the number N of satisfying assignments to F is equal to the number of cliques in G. Since any subset of the max clique is a clique, $N \geq 2^\omega$. Since each clique has size at most ω,

$$N \leq \binom{n}{\omega} + \binom{n}{\omega - 1} + \cdots + \binom{n}{0} \leq n^\omega.$$

Therefore, $\omega \leq \lg N \leq \omega \lg n$, so $\lg N/\lg n \leq \omega \leq \lg N$. Thus if $\lg N$ can be approximated to within a factor a, then ω can be approximated to within a factor $a \lg n$. Observing that the additional $\lg n$ factor is negligible in the proof of Theorem 2.13 completes the proof. □

As a corollary, using Valiant's reduction [21], we can show that computing the permanent of matrices with entries in $\{-1,0,1\}$ is hard.

COROLLARY 4.2. *If the log of the permanent of a matrix having positive permanent and entries in $\{-1,0,1\}$ can be approximated to within a factor of n^ϵ, then $NP = P$. If, for some constant a, approximating $\log^{(k+1)}$ of the permanent of a matrix having positive permanent and entries in $\{-1,0,1\}$ to within a factor a is in $FZPP_k$, then $NP_k = ZPP_k$.*

Proof. Valiant [21] showed that the number of satisfying assignments to a 3CNF formula, and hence a 2CNF formula, can be expressed as the permanent of a -1, 0, 1 matrix. □

Acknowledgments. I am grateful to Silvio Micali for several extremely useful discussions and to Laci Babai for observing that the conclusion $NP \subseteq BPP$ appearing in the preliminary version could be strengthened to $NP = ZPP$. I also thank Shafi Goldwasser and Mario Szegedy for helpful discussions and the anonymous referees for a careful reading and many valuable comments.

REFERENCES

[1] M. AJTAI, J. KOMLOS, AND E. SZEMEREDI, *Deterministic simulation in Logspace*, in Proc. 19th Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1987, pp. 132–140.

[2] N. ALON, U. FEIGE, A. WIGDERSON, AND D. ZUCKERMAN, *Derandomized graph products*, Comput. Complexity, 5 (1995), pp. 60–75.

[3] S. ARORA, C. LUND, R. MOTWANI, M. SUDAN, AND M. SZEGEDY, *Proof verification and intractibility of approximation problems*, in Proc. 33rd Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1992, pp. 14–23.

[4] S. ARORA AND S. SAFRA, *Approximating clique is NP-complete*, in Proc. 33rd Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1992, pp. 2–13.

[5] L. BABAI, L. FORTNOW, AND C. LUND, *Non-deterministic exponential time has two-prover interactive protocols*, Comput. Complexity, 1 (1991), pp. 16–25.

[6] M. Bellare, S. Goldwasser, C. Lund, and A. Russell, *Efficient probabilistically checkable proofs and applications to approximation*, in Proc. 25th Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1993, pp. 294–304.

[7] M. Bellare and M. Sudan, *Improved non-approximability results*, in Proc. 26th Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1994, pp. 184–193.

[8] P. Berman and G. Schnitger, *On the complexity of approximating the independent set problem*, Inform. and Comput., 96 (1992), pp. 77–94.

[9] A. Cohen and A. Wigderson, *Dispersers, deterministic amplification, and weak random sources*, in Proc. 30th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1989, pp. 14–19.

[10] S. A. Cook, *The complexity of theorem-proving procedures*, in Proc. 3rd Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1971, pp. 151–158.

[11] U. Feige, S. Goldwasser, L. Lovasz, S. Safra, and M. Szegedy, *Approximating clique is almost NP-complete*, in Proc. 32nd Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1991, pp. 2–12.

[12] 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.

[13] M. Jerrum and U. Vazirani, *A mildly exponential approximation algorithm for the permanent*, in Proc. 33rd Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1992, pp. 320–326.

[14] 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. 85–103.

[15] R. M. Karp, M. Luby, and N. Madras, *Monte-Carlo approximation algorithms for enumeration problems*, J. Algorithms, 10 (1989), pp. 429–448.

[16] C. Lund and M. Yannakakis, *On the hardness of approximating minimization problems*, in Proc. 25th Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1993, pp. 286–293.

[17] A. Panconesi and D. Ranjan, *Quantifiers and Approximation*, in Proc. 22nd Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1990, pp. 446–456.

[18] C. H. Papadimitriou and M. Yannakakis, *Optimization, approximation, and complexity classes*, J. Comput. System Sci., 43 (1991), pp. 425–440.

[19] M. Sántha, *On using deterministic functions to reduce randomness in probabilistic algorithms*, Inform. and Comput., 74 (1987), pp. 241–249.

[20] M. Sipser, *Expanders, randomness, or time versus space*, J. Comput. System Sci., 36 (1988), pp. 379–383.

[21] L. G. Valiant, *The complexity of computing the permanent*, Theoret. Comput. Sci., 8 (1979), pp. 189–201.

[22] D. Zuckerman, *Simulating BPP using a general weak random source*, Algorithmica, to appear; preliminary version in Proc. 32nd Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1991, pp. 79–89.

SIAM J. COMPUT.
Vol. 25, No. 6, pp. 1305–1317, December 1996

010

A LINEAR-TIME ALGORITHM FOR FINDING TREE-DECOMPOSITIONS OF SMALL TREEWIDTH*

HANS L. BODLAENDER†

Abstract. In this paper, we give for constant k a linear-time algorithm that, given a graph $G = (V, E)$, determines whether the treewidth of G is at most k and, if so, finds a tree-decomposition of G with treewidth at most k. A consequence is that every minor-closed class of graphs that does not contain all planar graphs has a linear-time recognition algorithm. Another consequence is that a similar result holds when we look instead for *path*-decompositions with pathwidth at most some constant k.

Key words. graph algorithms, treewidth, pathwidth, partial k-trees, graph minors

AMS subject classifications. 68R10, 05C85, 05C05

1. Introduction.

1.1. Background. The notions of "tree-decomposition" and "treewidth" have received much attention recently, not in the least due to the important role they play in the deep results on graph minors by Robertson and Seymour (see, e.g., [27, 28, 29, 30, 31] and many other papers in this series). (See also [21].) Also, many graph problems, including a very large number of well-known NP-hard problems, have been shown to be linear-time solvable on graphs that are given together with a tree-decomposition of treewidth at most k for constant k. (See, among other sources, [2, 5, 6, 7, 8, 12, 14, 15, 16, 33, 35].)

The first step of algorithms that exploit the small treewidth of input graphs is to find a tree-decomposition with treewidth bounded by a constant—although possibly not optimal. Thus far, this step has dominated the running time of most algorithms since the second step (some kind of "dynamic-programming" algorithm using the tree-decomposition) usually costs only linear time. The best algorithm known so far for this "first step" was an algorithm by Reed [26], which costs $O(n \log n)$. In this paper, we improve on this result and give a linear-time algorithm.

The problem "Given a graph $G = (V, E)$ and an integer k, is the treewidth of G at most k?" is NP-complete [3]. Much work has been done on this problem for constant k. For $k = 1, 2, 3$, linear-time algorithms exist [25]. Recently, Sanders [32] established a complex linear-time algorithm for the case where $k = 4$. Arnborg et al. [3] showed that the problem is solvable in $O(n^{k+2})$ time for constant k. Then Robertson and Seymour gave a nonconstructive proof of the existence of $O(n^2)$ decision algorithms [31]. Actually, this algorithm is of a "two-step" form, as described above. The first step is to apply an $O(n^2)$ algorithm that either outputs that the treewidth of G is larger than k or outputs a tree-decomposition with width at most $4k$. (Actually, the result is stated in [31] in terms of "branchwidth," but this is an unimportant technical difference.) The second step uses the notion of graph minors. A graph G is a minor of a graph H if G can be obtained from H by a series of vertex deletions, edge deletions, and edge contractions. Robertson and Seymour have shown that every class of graphs $\mathcal{G}$ that is closed under the taking of minors has a finite set of graphs, called

* Received by the editors June 28, 1993; accepted for publication (in revised form) March 15, 1995. This research was partially supported by the EC ESPRIT Basic Research Actions contract 7141 (project ALCOM II).

† Department of Computer Science, Utrecht University, P. O. Box 80.089, 3508 TB Utrecht, the Netherlands (hansb@cs.ruu.nl).

the obstruction set, with the property that a graph belongs to $\mathcal{G}$ if and only if it has no graph from the obstruction set as a minor. Since the class of graphs with treewidth at most k is closed under minors for every fixed value k, a finite characterization in terms of forbidden minors exists for this class. Hence the second step of the algorithm checks whether this characterization holds for the input graph. This step can be done in linear time using dynamic-programming techniques as used, e.g., in [5, 6, 14]. In [9] (using results from [20]), it was shown that the nonconstructive elements can be avoided using self-reduction without increasing the running time by more than a (huge) constant factor.

Both Lagergren [23] and Reed [26] improved on the "first step." Lagergren gave a sequential algorithm that uses $O(n \log^2 n)$ time and a parallel algorithm that uses $O(n)$ processors and $O(\log^3 n)$ time. Reed gave a sequential $O(n \log n)$ algorithm that has a parallel implementation with $O(n/\log n)$ processors and $O(\log^2 n)$ time. A related probabilistic result (with running time $O(n \log^2 n + n|\log p|)$, where p is the probability of error) was found by Matousěk and Thomas [25]. Each of these algorithms either determines that the input graph G has treewidth greater than k or finds a tree-decomposition of G with treewidth bounded by some constant (linear in k). They all are based upon finding "balanced separators" in some clever way. Our algorithm uses a different approach: we reduce the problem in linear time to a problem on a smaller graph by edge contraction or by removing "simplicial vertices."

Independently, Lagergren and Arnborg [24] and Bodlaender and Kloks [12, 22] showed that the "second step" can be done without the use of graph minors and gave explicit algorithms to test in linear time whether G has treewidth at most k once a tree-decomposition of G with bounded treewidth is available. Moreover, from these results, it follows that a technique of Fellows and Langston [19] can be used to compute the obstruction set of the class of graphs with treewidth $\leq k$. Bodlaender and Kloks also showed how, if it exists, a tree-decomposition with width at most k can be computed in the same time bounds. Results of a similar flavor were obtained independently by Abrahamson and Fellows [1].

Recognition algorithms for graphs with treewidth $\leq k$ (k constant) have been designed by Arnborg et al. [4]. These algorithms use linear time but polynomial—not linear—memory. (It is allowed that the algorithm consults the contents of memory that is never written to.) A disadvantage of this approach is that it is not known how to *construct* tree-decompositions with small treewidth by the method.

1.2. Main idea of algorithm.

The main result in this paper is the following.

THEOREM 1.1. *For all $k \in N$, there exists a linear-time algorithm that tests whether a given graph $G = (V, E)$ has treewidth at most k and, if so, outputs a tree-decomposition of G with treewidth at most k.*

We now give an outline of how this result is obtained.

We begin by introducing some notation. For a value d to be fixed later, we define *low-degree* vertices as vertices of degree at most d and *high-degree* vertices as vertices of degree greater than d. A vertex is *friendly* if it is a low-degree vertex and adjacent to another low-degree vertex. A vertex is *simplicial* if its neighbors form a clique. The *improved* graph of a graph G is obtained by adding edges between all vertices that have at least $k+1$ common neighbors of degree at most k. A vertex is *I-simplicial* in a graph G if it is simplicial in the improved graph of G and has degree at most k in G.

The algorithm distinguishes between two cases:

1. *There are "many" friendly vertices.* As shown in §3, any maximal matching

in G contains in this case "sufficiently many" ($\Omega(n)$) edges. We compute the graph G' obtained by contracting all edges in a maximal matching. Recursively, we compute a tree-decomposition of treewidth at most k of G' or conclude that the treewidth of G' and hence the treewidth of G is larger than k. From this tree-decomposition, we can easily build a tree-decomposition of G with treewidth at most $2k+1$. This latter tree-decomposition is used to solve the problem using the algorithm of Bodlaender and Kloks [12, 22]: using the tree-decomposition of G with treewidth at most $2k+1$, it decides whether the treewidth of G is at most k and, if so, finds a tree-decomposition of G with treewidth at most k.

2. *G has "only few" friendly vertices.* In this case, the algorithm starts by computing the improved graph of G. In §4, it is shown that this improved graph has treewidth at most k if and only if G has treewidth at most k. Also, in §4, it is shown that in this case, the improved graph of G has "sufficiently many" ($\Omega(n)$) vertices that are I-simplicial (unless the treewidth of G is more than k). Recursively, a tree-decomposition with treewidth at most k is computed of the graph G' obtained by removing all I-simplicial vertices from the improved graph of G, or we conclude that the treewidth of G' and hence of G is larger than k. Given such a tree-decomposition of G', a tree-decomposition of G with treewidth at most k is computed as follows: since the neighbors of an I-simplicial vertex v form a clique in G', a well-known lemma tells us that there is one node i in the tree-decomposition of G' with X_i containing all neighbors of v. Then we add a new node to the tree-decomposition, adjacent to i, containing v and its neighbors. In this way, we obtain a tree-decomposition of G with treewidth at most k.

In each case, the amount of work of the nonrecursive steps is linear, and each G' has size at most a constant fraction of the size of G. It follows that the algorithm uses linear time.

The basic algorithm will be given in §5. Some implementation details will be discussed in §6. Finally, some consequences of the result will be discussed in §7.

2. Definitions and preliminary results.

The notion of treewidth was introduced by Robertson and Seymour [27].

DEFINITION. *A* tree-decomposition of a graph $G = (V, E)$ *is a pair* (X, T), *where* $T = (I, F)$ *is a tree and* $X = \{X_i \mid i \in I\}$ *is a family of subsets of* V, *one for each node of* T, *such that*

(i) $\bigcup_{i \in I} X_i = V$,

(ii) *for all edges* $(v, w) \in E$, *there exists an* $i \in I$ *with* $v \in X_i$ *and* $w \in X_i$, *and*

(iii) *for all* $i, j, k \in I$, *if* j *is on the path from* i *to* k *in* T, *then* $X_i \cap X_k \subseteq X_j$.

The treewidth of a tree-decomposition $(\{X_i \mid i \in I\}, T = (I, F))$ *is* $\max_{i \in I} |X_i| - 1$. *The* treewidth of a graph G *is the minimum treewidth over all possible tree-decompositions of* G.

There are several equivalent notations, e.g., a graph is a partial k-tree if and only if its treewidth is at most k [34].

LEMMA 2.1. (See, e.g., [13].) *Suppose* $(\{X_i \mid i \in I\}, T = (I, F))$ *is a tree-decomposition of* $G = (V, E)$.

(i) *If* $W \subseteq V$ *forms a clique in* G, *then there exists an* $i \in I$ *with* $W \subseteq X_i$.

(ii) *If each vertex in* $W_1 \subseteq V$ *is adjacent to each vertex in* $W_2 \subseteq V$, *then there exists an* $i \in I$ *with* $W_1 \subseteq X_i$ *or* $W_2 \subseteq X_i$.

The *contraction* operation removes two adjacent vertices v and w and replaces them with one new vertex that is made adjacent to all vertices that were adjacent to v and w.

We say that a tree-decomposition (X, T) of treewidth k is *smooth* if for all $i \in I, |X_i| = k+1$ and for all $(i,j) \in F, |X_i \cap X_j| = k$. Any tree-decomposition of a graph G can be transformed to a smooth tree-decomposition of G with the same treewidth. Apply the following operations until none is possible:

(i) If for $(i,j) \in F$, $X_i \subseteq X_j$, then contract the edge (i,j) in T and take as the new node $X_{j'} = X_j$.

(ii) If for $(i,j) \in F$, $X_i \not\subseteq X_j$ and $|X_j| < k+1$, then choose a vertex $v \in X_i - X_j$ and add v to X_j.

(iii) If for $(i,j) \in F$, $|X_i| = |X_j| = k+1$ and $|X_i - X_j| > 1$, then subdivide the edge (i,j) in T; let i' be the new node; choose a vertex $v \in X_i - X_j$ and a vertex $w \in X_j - X_i$, and let $X_{i'} = X_i - \{v\} \cup \{w\}$.

LEMMA 2.2. *If (X,T) is a smooth tree-decomposition of $G = (V,E)$ with treewidth k, then $|I| = |V| - k$.*

Proof. The proof is by induction on $|I|$. If $|I| = 1$, then clearly $|V| = k+1$. Suppose that the lemma holds for $|I| = r-1$. Consider a smooth tree-decomposition (X,T) of a graph $G = (V,E)$ with treewidth k and $|I| = r$. Let i be a leaf of T. There is a unique vertex v that belongs to X_i but not to any set X_j, $j \in I - \{i\}$. If we remove node i from T, we get a smooth tree-decomposition of $G[V - \{v\}]$ with treewidth k and with $|I| - 1$ nodes. The result now follows by induction. □

The following well-known lemma can be easily proved by induction on the number of vertices, removing vertices as in Lemma 2.2.

LEMMA 2.3. *If the treewidth of $G = (V,E)$ is at most k, then $|E| \leq k|V| - \frac{1}{2}k(k+1)$.*

The set of neighbors of a vertex v in $G = (V,E)$ is denoted by $N_G(v) = \{w \in V \mid (v,w) \in E\}$.

THEOREM 2.4. (See Bodlaender and Kloks [12, 22].) *For all k and l, there exists a linear-time algorithm that, when given a graph $G = (V,E)$ together with a tree-decomposition (X,T) of G with treewidth at most l, determines whether the treewidth of G is at most k and, if so, finds a tree-decomposition of G with treewidth at most k.*

Analysis of this algorithm shows that its constant factor is at most $l^{l-2} \cdot ((2l+3)^{2l+3} \cdot (\frac{8}{3} \cdot 2^{2k+2})^{2l+3})^{2l-1}$, i.e., when $l = O(k)$, exponential in k^3. The analysis leading to this constant is rather crude, however, and a precise analysis should give a much better and smaller estimate.

3. Friendly, high-degree, and low-degree vertices.

In this section, we introduce the concepts of the "friendly," "high-degree," and "low-degree" vertex. We show that a graph with treewidth at most k has "few" high-degree vertices, and when it has "many" friendly vertices, then it has a "large" maximal matching.

In the remainder, we assume that k is a given fixed constant. Let

$$c_1 = \frac{1}{k^2 \cdot (k+1) \cdot (4k^2 + 12k + 16)}$$

and

$$c_2 = \frac{1}{8k^2 + 24k + 32}.$$

Note that

$$c_2 = \frac{1}{4k^2 + 12k + 16} - \frac{c_1 \cdot k^2 \cdot (k+1)}{2}.$$

Let $d = 2k^3 \cdot (k+1) \cdot (4k^2 + 12k + 16)$. Note that $d = 2k/c_1$. We say that a vertex with degree at most d is a *low-degree vertex* and a vertex with degree larger than d is a *high-degree vertex.* A vertex is said to be *friendly* if it is a low-degree vertex and is adjacent to at least one other low-degree vertex.

We show below that c_1 is an upper bound on the fraction of high-degree vertices. In this section and the next, we show that c_2 is a lower bound on the fraction of vertices that can be removed in one of the two cases, as mentioned in §1.2.

LEMMA 3.1. *There are fewer than $c_1 \cdot |V|$ high-degree vertices in a graph with treewidth k.*

Proof. If there are n_l high-degree vertices, then G must contain at least $n_l \cdot d/2$ edges. By Lemma 2.3, $n_l \cdot d/2 < k|V|$. □

A *maximal matching* of a graph $G = (V, E)$ is a set of edges $M \subseteq E$ such that no two edges in M share an endpoint and every $e \in E - M$ shares an endpoint with an edge in M. We can easily find a maximal matching in $O(|V| + |E|)$ time with a greedy algorithm. Note that by Lemma 2.3, $O(|V| + |E|) = O(|V|)$ for graphs $G = (V, E)$ with their treewidth bounded by a constant.

LEMMA 3.2. *If there are n_f friendly vertices in $G = (V, E)$, then any maximal matching of G contains at least $n_f/(2d)$ edges.*

Proof. Consider a maximal matching M. Any friendly vertex must be endpoint of an edge in M or adjacent to a friendly vertex that is an endpoint of an edge in M. With each edge e of M, we associate the at most $2d$ friendly vertices that are endpoints of e or adjacent to friendly (and hence low-degree) endpoints of e. If a friendly vertex has not been associated with at least one edge in M, then M is not maximal. Hence $|M| \geq n_f/(2d)$. □

Let M be a maximal matching in $G = (V, E)$, and let $G' = (V', E')$ be the graph obtained by contracting all edges in M. Define $f_M : V \to V'$ by $f_M(v) = v$ if v is not an endpoint of an edge in M, and let $f_M(v) = f_M(w)$ be the vertex that the contraction of the edge $(v, w) \in M$ results in.

LEMMA 3.3. *Let M, G, G', and f_M be as above. If (X, T) is a tree-decomposition of G' with treewidth k, then (Y, T) defined by $Y_i = \{v \in V \mid f_M(v) \in X_i\}$ is a tree-decomposition of G with treewidth at most $2k + 1$.*

Proof. This easily follows from the definitions. □

LEMMA 3.4. (See, e.g., [27].) *If G' is a minor of G, then the treewidth of G' is at most the treewidth of G.*

4. Simplicial vertices.

In this section, we introduce the improved graph of a graph G. We show that a graph G has treewidth at most k if and only if its improved graph has treewidth at most k. The main result of this section is Theorem 4.3, which states that every graph of treewidth at most k contains "many" friendly vertices or "many" I-simplicial vertices.

For a graph $G = (V, E)$, let the *improved* graph $G' = (V, E')$ of G be the graph obtained by adding an edge (v, w) to E for all pairs v, $w \in V$ such that v and w have at least $k + 1$ common neighbors of degree at most k in G.

LEMMA 4.1. *If the treewidth of G is at most k, then the treewidth of the improved graph of G is at most k. Moreover, any tree-decomposition of G with treewidth at most k is also a tree-decomposition of the improved graph with treewidth at most k, and vice versa.*

Proof. Suppose that (X, T) is a tree-decomposition of $G = (V, E)$ with treewidth at most k. Consider vertices v and w with at least $k + 1$ common neighbors. By Lemma 2.1(ii), there exists either an $i \in I$ with $v, w \in X_i$ or an $i \in I$ with X_i

containing the set W of all common neighbors of v and w. In the latter case, (X, T) is also a tree-decomposition of the graph G'' obtained from G by adding edges between all vertices in W. However, G'' contains a clique with at least $k+2$ vertices (namely, $W \cup \{v\}$) and has treewidth at most k. This contradicts Lemma 2.1(i).

Therefore, for all v and w that have $k+1$ common neighbors, there exists an $i \in I$ with $v, w \in X_i$. Hence (X, T) is also a tree-decomposition of the improved graph of G. The lemma now follows directly. □

We say that a vertex v is *simplicial in G* if its neighbors form a clique in G. We say that v is I-simplicial if it is simplicial in the improved graph of G and is of degree at most k in G.

We now derive via a series of lemmas the following result, which states that if we have "few" friendly vertices and the treewidth of G is at most k, then we have "many" I-simplicial vertices.

THEOREM 4.2. *For every graph $G = (V, E)$ with treewidth at most k, at least one of the following properties holds:*

(i) *G contains at least $|V|/(4k^2 + 12k + 16)$ friendly vertices.*

(ii) *The improved graph of G contains at least $c_2|V|$ I-simplicial vertices.*

Proof. The proof of Theorem 4.2 will be given with help of several lemmas.

A vertex $v \in V$ is said to be *T-simplicial* with respect to some tree-decomposition (X, T) if it is not friendly and there exists a node $i \in I$ such that all neighbors of v belong to X_i. A T-simplicial vertex has degree at most k since all of its neighbors belong to a set X_i, $|X_i| \leq k+1$.

LEMMA 4.3. *For all smooth tree-decompositions (X, T) of $G = (V, E)$ with treewidth k, the following conditions hold:*

(i) *We can associate with every leaf i of T a low-degree vertex $v \in X_i$ that is friendly or T-simplicial with respect to (X, T), and there does not exist a $j \in I$, $j \neq i$, with $v \in X_j$.*

(ii) *We can associate with every path $i_0, i_1, \ldots, i_{k^2+3k+3}$ in T with $i_1, \ldots, i_{k^2+3k+2}$ nodes of degree 2 in T at least one vertex $v \in X_{i_1} \cup \cdots \cup X_{i_{k^2+3k+2}}$ that is friendly or T-simplicial with respect to (X, T) such that v does not belong to a set X_j with $j \in I$ a node not on this path.*

Proof. (i) Let j be the neighbor of leaf i in T. Let v be the unique vertex in $X_i - X_j$. v is adjacent to only vertices in X_i. Either all neighbors of v are of high degree, in which case v is T-simplicial with respect to (X, T), or a neighbor of v is of low degree, in which case v is friendly.

(ii) Note that $|X_{i_0} \cup \cdots \cup X_{i_{k^2+3k+3}}| = k^2 + 4k + 4 \leq d$. Hence all vertices in $X_{i_1} \cup \cdots \cup X_{i_{k^2+3k+2}} - (X_{i_0} \cup X_{i_{k^2+3k+3}})$ are of low degree. Suppose that neither of them is friendly, i.e., they are adjacent to only high-degree vertices in $X_{i_0} \cup X_{i_{k^2+3k+3}}$. Suppose that X_{i_0} contains r high degree vertices, say $w_1, \ldots, w_r$. Clearly, $r \leq |X_{i_0}| = k+1$. For each s, where $1 \leq s \leq r$, assume that w_s belongs to successive sets $X_{i_0}, X_{i_1}, \ldots, X_{i_{w_s}}$. Suppose w.l.o.g. $i_{w_1} \leq i_{w_2} \leq \cdots \leq i_{w_r}$. If some low-degree vertex v belongs to exactly one set X_{i_j}, $1 \leq j \leq k^2 + 3k + 2$, then it must be T-simplicial with respect to (X, T). If some low-degree vertex v belongs only to (a subset of) sets $X_{i_{w_j+1}}, \ldots, X_{i_{w_{j+1}}}$, then all neighbors of v belong to $X_{i_{w_{j+1}}}$; hence v is T-simplicial with respect to (X, T). All vertices in $X_{i_1} \cup \cdots \cup X_{i_{k^2+3k+2}}$ that are not of one of these two types must belong to at least one of the sets $X_{i_0}, X_{i_{w_1}}, \ldots, X_{i_{w_r}}, X_{i_{k^2+3k+3}}$. These are, in total, at most $(k+1)(k+3) = k^2 + 4k + 3$ vertices. Therefore, at least one vertex in $X_{i_1} \cup \cdots \cup X_{i_{k^2+3k+2}} - X_{i_0} - X_{i_{k^2+3k+3}}$ must be T-simplicial with respect to (X, T). □

A leaf-path collection of a tree T is a collection of leaves in T plus a collection of paths of length k^2+3k+4 in T where all nodes on a path that are not endpoints of a path have degree 2 in T and do not belong to any other path in the collection. The size of the collection is the total number of leaves plus the total number of paths in the collection.

LEMMA 4.4. *Each tree with r nodes contains a leaf-path collection of size at least $r/(2k^2+6k+8)$.*

proof Let r_b be the number of nodes of degree at least 3, r_l be the number of leaves, and r_2 be the number of nodes of degree 2. Clearly, $r_b < r_l$. All nodes of degree 2 belong to $< r_l + r_b$ connected components of the forest, obtained by removing all leaves and all nodes with degree 3 or larger from the tree. Each such component contains at most k^2+3k+3 nodes that are not part of a leaf-path collection of maximum size. Therefore, there are fewer than $(r_b+r_l)(k^2+3k+3)$ nodes of degree 2 that are not on a path in the collection. Hence there are at least

$$\frac{r_2-(r_b+r_l)(k^2+3k+3)}{k^2+3k+4}$$

paths in a leaf-path collection of maximum size. It follows that the maximum size of a leaf-path collection is at least

$$\max\left(r_l, \frac{r_2-(r_b+r_l)(k^2+3k+3)}{k^2+3k+4}+r_l\right) \geq \frac{1}{2}\cdot\frac{r}{k^2+3k+4}. \qquad \square$$

COROLLARY 4.5. *If (X,T) is a smooth tree-decomposition of $G=(V,T)$ with treewidth k, then G contains at least $|V|/(2k^2+6k+8)-1$ vertices that are friendly or T-simplicial with respect to (X,T).*

Proof. T contains $|V|-k$ nodes (Lemma 2.2). Now apply Lemmas 4.3 and 4.4. □

A set $Y \subseteq V$ of high-degree vertices is said to be *semiimportant* with respect to the tree-decomposition (X,T) of $G=(V,E)$ if there exists an $i \in I$ with $Y \subseteq X_i$. A set Y is said to be *important* if it is semiimportant with respect to (X,T) and not contained in any larger semiimportant set with respect to (X,T).

LEMMA 4.6. *Let (X,T) be a tree-decomposition of $G=(V,E)$ with treewidth k. The number of different important sets with respect to (X,T) is at most the number of high-degree vertices in G.*

Proof. Let L be the set of high-degree vertices in G. $(\{X_i \cap L \mid i \in I\}, T)$ is a tree-decomposition of $G[L]$. Each important set Y is a set $X_i \cap L$ that is not contained in another set $X_{i'} \cap L$. Repeatedly contract edges (i,i') in T with $X_i \cap L \supseteq X_{i'} \cap L$ with the newly formed node containing all vertices in X_i. The resulting tree-decomposition of $G[L]$ contains the same maximal sets X_i and will have at most $|L|$ nodes. □

A function f that maps each T-simplicial (with respect to some tree-decomposition (X,T)) vertex v to an important (with respect to (X,T)) set Y with $N_G(v) \subseteq Y$ is called a *T-simplicial-to-important function* for (X,T). By definition, a T-simplicial-to-important function always exists.

LEMMA 4.7. *Let f be a T-simplicial-to-important function for a smooth tree-decomposition (X,T) of $G=(V,E)$ with treewidth k. Let Y be an important set with respect to (X,T). Then at most $\frac{1}{2}k^2(k+1)$ T-simplicial vertices with respect to (X,T) (and G) in $f^{-1}(Y)$ are not I-simplicial.*

Proof. Assign each non-I-simplicial T-simplicial vertex v to a pair of neighbors of v, that are nonadjacent in the improved graph. To each pair of vertices, there cannot be assigned more than k vertices since otherwise they would have at least $k+1$ common neighbors of degree at most k and there would be an edge between them in the improved graph.

It follows that the number of non-I-simplicial T-simplicial vertices v with $f(v) = Y$ is at most $\frac{1}{2}|Y| \cdot (|Y| - 1) \leq \frac{1}{2}k^2(k+1)$. □

We can now prove Theorem 4.2. Suppose that G contains fewer than $|V|/(4k^2 + 12k + 16)$ friendly vertices and that the treewidth of G is at most k. By Lemma 3.1, there are at most $c_1 \cdot |V|$ high-degree vertices in G, and hence, by Lemma 4.6, the number of important sets with respect to an arbitrary smooth tree-decomposition (X,T) of G with treewidth $\leq k$ is at most $c_1 \cdot |V|$. Using both the fact that a T-simplicial-to-important function always exists and Lemma 4.7, it follows that at most $\frac{1}{2}k^2(k+1) \cdot (c_1 \cdot |V| - 1)$ T-simplicial vertices with respect to (X,T) are not I-simplicial. Using Corollary 4.5, it follows that $|V|/(2k^2 + 6k + 8) - 1 - |V|/(4k^2 + 12k + 16) - \frac{1}{2}k^2(k+1) \cdot (c_1 \cdot |V| - 1) \geq c_2 \cdot |V|$ vertices are I-simplicial. This completes the proof of Theorem 4.2. □

LEMMA 4.8. *Let (X,T) be a tree-decomposition of treewidth at most k of the graph G' obtained by removing all I-simplicial vertices (and adjacent edges) from the improved graph of graph $G = (V,E)$. Then for all I-simplicial vertices v, there exists an $i \in I$ with $N_G(v) \subseteq X_i$.*

Proof. Note that, by definition, I-simplicial vertices are nonadjacent in G, and their neighborhood forms a clique in the improved graph of G. The result now follows directly from Lemma 2.1(i). □

5. Main algorithm.

We now give a recursive description of our main algorithm. Some details will be discussed in §6. Our algorithm, when given a graph $G = (V,E)$, either

(i) outputs that the treewidth of G is larger than k or

(ii) outputs a tree-decomposition of G with treewidth at most k.

For "very small graphs" (i.e., with at most some constant number of vertices), any other finite algorithm is used to solve the problem. Otherwise, the following algorithm is used:

First, check whether $|E| \leq k \cdot |V| - \frac{1}{2}k(k+1)$. If this is not the case, we know by Lemma 2.3 that the treewidth of G is larger than k: **stop**.

Now count the number of friendly vertices. If there are at least $|V|/(4k^2+12k+16)$ friendly vertices, do the following:

(i) Find a maximal matching $M \subseteq E$ in G.

(ii) Compute the graph $G' = (V', E')$ obtained by contracting every edge in M.

(iii) Recursively apply the algorithm to G'.

(iv) If G' has treewidth larger than k, **stop**. The treewidth of G is also larger than k. (See Lemma 3.4.)

(v) Suppose that the recursive call yielded a tree-decomposition (X,T) of G' with treewidth k. Construct a tree-decomposition (Y,T) of G with treewidth at most $2k+1$, as in Lemma 3.3.

(vi) Use the algorithm of Theorem 2.4 to compute whether the treewidth of G is at most k and, if so, compute a tree-decomposition of G of treewidth at most k.

If there are fewer than $|V|/(4k^2 + 12k + 16)$ friendly vertices, do the following:

(i) Compute the improved graph of G. (See §6.)

(ii) If there exists an I-simplicial vertex with degree at least $k+1$, then **stop**: the improved graph of G contains a clique with $k+2$ vertices; hence the treewidth of G is more than k.

(iii) Put all I-simplicial vertices in some set SL. Compute the graph G' obtained by removing all I-simplicial vertices and adjacent edges from G.

(iv) If $|SL| < c_2|V|$, then **stop**: the treewidth of G is larger than k. (See Theorem 4.2.)

(v) (Now $|SL| \geq c_2|V|$.) Recursively apply the algorithm on G'.

(vi) If the treewidth of G' is larger than k, then **stop**: since G' is a subgraph of G, we also have that the treewidth of G is larger than k.

(vii) Suppose that the recursive call yielded a tree-decomposition (X, T) of G' with treewidth k. For all $v \in SL$, find an $i_v \in V$ with $N_G(v) \subseteq X_{i_v}$, add a new node j_v to T with $X_{j_v} = \{v\} \cup N_G(v)$, and make j_v adjacent to i_v in T. (Such a node i_v exists by Lemma 4.8.) The result is a tree-decomposition of G with treewidth at most k.

The correctness of the algorithm follows from results given in §§2 and 4.

The running time of the algorithm can be estimated as follows. We recursively apply the algorithm on either a graph with $(1 - 1/(2d(4k^2 + 12k + 16))) \cdot |V|$ vertices (Lemma 3.2) or a graph with $(1 - c_2)|V|$ vertices. Write

$$c_3 = \frac{1}{8k^6 + 32k^4 + 56k^3 + 32k^3} = \max\left(1 - c_2,\ 1 - \frac{1}{2d \cdot (4k^2 + 12k + 16)}\right).$$

Since all nonrecursive steps have a linear-time implementation (see also §6), we have that if the algorithm takes $T(n)$ time on a graph with n vertices in the worst case, then $T(n) \leq T(c_3 \cdot n) + O(n)$; hence $T(n) = O(n)$. It also follows that the algorithm uses linear memory.

6. Some details of the algorithm.

In this section, we show that the steps of the algorithm given in §5 can be implemented in linear time and linear memory. Most steps are either rather straightforward and thus left to the reader or follow from earlier results. Note that we may always assume that the number of edges that we are working with is linear in the number of vertices. All graphs that we work with will be represented by their adjacency lists.

When we contract the edges in a matching M, we directly get an implicit representation of G' by a bag of edges, where some edges of G' may appear twice. By bucket sorting this bag of edges twice, we can remove all multiple copies of edges and easily obtain an adjacency list representation of G'.

Computing the improved graph and the I-simplicial vertices. Number the vertices $v_1, v_2, \ldots, v_n$. We use a queue Q that contains triples of the form $((v, w), x)$ with $v, w, x \in V$ or of the form $((v, w), —)$, $v, w \in V$. Also, we use an array S with, for each $v_i \in V$, a list $S[v_i]$ containing pairs of vertices. For all $(v_i, v_j) \in E$ with $i < j$, put $((v_i, v_j), —)$ on Q. For all vertices $v \in V$ with degree at most k, for all pairs of neighbors $v_i, v_j \in N_G(v)$ with $i < j$, put $((v_i, v_j), v)$ on Q. Now "bucket sort" Q twice, once to the first-vertex entries and once to the second-vertex entries. After this double bucket sort, all pairs of the form $((v_i, v_j), \ldots)$ for fixed v_i and v_j will be in consecutive positions in Q. By inspecting Q, we can directly see what pairs of vertices have at least $k + 1$ common neighbors of degree at most k. (If at least $k + 1$ entries $((v_i, v_j), v)$ are adjacent in Q for some pair v_i, v_j, (v_i, v_j) must be present in the improved graph.) For each such pair (v_i, v_j) and if a triple $((v_i, v_j), —)$ is in Q, add the pair (v_i, v_j) to all lists $S[v]$ for vertices v with $((v_i, v_j), v)$ in Q. This all can be done in linear time using the consecutiveness of the pairs of the form $((v_i, v_j), \ldots)$.

Checking whether a vertex v of degree at most k is I-simplicial can be done by inspecting $S[v]$. $S[v]$ will consist precisely of all edges between neighbors of v. Since v has degree at most k, $S[v]$ is of size, bounded by a constant.

Adding I-simplicial vertices back in the tree-decomposition. Suppose that we have a tree-decomposition (X, T) of $G[V - SL]$ and we want to add all I-simplicial vertices in SL. For all $l \leq k$, we take a queue Q_l, in which we place all pairs $((v_{i_1}, \ldots, v_{i_l}), i)$ for $v_{i_1}, \ldots, v_{i_l} \in X_i$, $i \in I$, $i_1 < i_2 < \cdots < i_l$, and all pairs $((v_{i_1}, \ldots, v_{i_l}), v)$ with v I-simplicial and $N_G(v) = \{v_{i_1}, \ldots, v_{i_l}, i_1 < i_2 < \cdots < i_l\}$.

For each l, $1 \leq l \leq k$, bucket sort Q_l l times, once for each of the l positions in the l-tuple. All entries of the form $((v_{i_1}, \ldots, v_{i_l}), \ldots)$ will be at successive positions in Q_l after this operation. By a simple scan of Q_l, we can find for each entry $((v_{i_1}, \ldots, v_{i_l}), v)$ an entry of the form $((v_{i_1}, \ldots, v_{i_l}), i)$ for some $i \in I$. This node i is a node that the new node j_v with $X_{j_v} = \{v\} \cup N_G(v)$ can be made adjacent to.

Analysis of the constant factor. We now analyze the constant factor of the algorithm somewhat more precisely. (In the following analysis, k is no longer considered a constant.) There are two nonrecursive steps that take time with a constant factor that is not polynomial in k: the application of the algorithm of Theorem 2.4 and the addition of I-simplicial vertices back in the tree-decomposition. One directly sees that the former constant is largest. Note that Theorem 2.4 is applied with $l = 2k + 1$. Now, since $1 - c_3 = \Theta(k^{-5})$, $T(n) \leq T(c_3 n) + (2k+1)^{2k+1-2} \cdot ((2(2k+1)+3)^{2(2k+1)+3} \cdot (\frac{8}{3} \cdot 2^{2k+2})^{2(2k+1)+3})^{2(2k+1)-1} \cdot n$, we have that $T(n) = O(k^5 \cdot (2k+1)^{(2k+1)-2} \cdot ((2(2k+1)+3)^{2(2k+1)+3} \cdot (\frac{8}{3} \cdot 2^{2k+2})^{2(2k+1)+3})^{2(2k+1)-1} \cdot n)$, i.e., linear with a constant factor that is exponential in k^3.

7. Final remarks. A consequence of the result in this paper is that all results that state that certain problems are solvable in linear time for graphs that are given together with a tree-decomposition of width bounded by a constant are turned into results that state that these problems can be solved in linear time on graphs with treewidth bounded by a constant. One of the most notable of such results is the following.

THEOREM 7.1. *Every class of graphs that does not contain all planar graphs and is closed under the taking of minors has a linear-time recognition algorithm.*

Proof. See, e.g., [31]. Use the algorithm described in this paper to find a tree-decomposition of the input graph with treewidth bounded by a constant, and use this tree-decomposition to test for minor inclusion for all graphs in the obstruction set of the class. □

Note that the result of Theorem 7.1 is nonconstructive: it relies on the nonconstructively proven fact that every minor-closed class of graphs has a finite characterization in terms of an obstruction set (see [31]). Thus we know that an algorithm exists, but the algorithm itself is not known. Even worse, even if an obstruction set and hence the algorithm were known for a certain class of graphs, the algorithm obtained with this method would only produce "yes" and "no" answers and would not construct any additional desired information. (For example, the result states that the class of graphs which are subgraphs of a planar graph with diameter at most d is linear-time recognizable for fixed d. However, such an algorithm would not produce such planar supergraphs of diameter at most d for "yes" instances.) In [17, 18], several classes of graphs to which Theorem 7.1 can be applied can be found. For several of these classes, we expect that constructive linear-time algorithms for recognition and construction of solutions can be found. Recent research [10] shows that linear-time algorithms can be constructed that solve minimum-cut linear arrangement, search number, and some related problems for constant k and, for "yes" instances, output the required linear arrangement.

Note that the result shown in this paper is equivalent to stating that (for fixed k) partial k-trees can be recognized and embedded in a k-tree (or a chordal graph with maximum clique size $k+1$) in linear time. Also, a direct consequence is the following.

THEOREM 7.2. *For all $k \in N$, there exists a linear-time algorithm that tests whether a given graph $G = (V, E)$ has pathwidth at most k and, if so, outputs a path-decomposition of G with pathwidth at most k.*

Proof. First, use the algorithm described in this paper. When the treewidth of G is larger than k, then clearly we also have that the pathwidth of G is larger than k. Otherwise, use the result from [12, 22] that states that for all constants k and l, there exists a linear-time algorithm that, when given a graph G and a tree-decomposition of G with treewidth at most l, decides whether G has pathwidth at most k and, if so, outputs a path-decomposition of G with pathwidth at most k. □

The constant factor of the algorithm as derived above is very large—much too large for practical purposes. We remark that the analysis used some crude arguments, and it can be expected that the real constant factor of the algorithm is much smaller. However, the algorithm in its present form is probably not practical, even for $k = 4$. An interesting topic for further research is the development of a practical algorithm for the "treewidth $\leq k$" problem. Ideas and techniques in this paper may help to develop such algorithms. For instance, finding I-simplicial vertices is done quite fast and may be a good heuristic.

It is also possible to modify the algorithm such that it uses the algorithm in [12, 22] only on tree-decompositions with treewidth at most $k+1$ at the cost of increasing the running time to $O(n \log n)$. Provided that the algorithm in [12, 22] can be made fast enough, this modification may well be quite practical for small values of k (like $k = 4$ or $k = 5$). The idea is as follows: instead of using the construction of Lemma 3.3, first find a set M' of at least $|M|/(k+1)$ edges in M such that no two vertices that are the result of contracting an edge in M' belong to a common set X_i. (Such a set can be quickly found in $O(n)$ time: the graph H obtained by adding an edge between every pair of vertices in G' that share a common set X_i is a graph with treewidth k and hence is $(k+1)$-colorable. Hence the set of vertices that are a result of an edge contraction contains an independent set in H of size at least $|M|/(k+1)$. Take M', the set of edges corresponding to the vertices in this independent set.) Define $f_{M'}$ as in §2. Now (Y, T) defined by $Y_i = \{v \in v \mid f_{M'} \in X_i\}$ is a tree-decomposition of the graph obtained from G by contracting all edges in M' with treewidth at most $k+1$. Use the algorithm from [12, 22] to find a tree-decomposition of treewidth at most k of this graph. Repeat the process with this last tree-decomposition and edge set $M - M'$ until the edge set is empty. These are at most $O(\log n)$ iterations. (This observation was also made by Jens Lagergren.)

It is possible to implement the algorithm such that it runs on a pointer machine (a correct use of pointers is necessary such that the addressing in the bucket-sort algorithms can be done) and still uses linear time. We omit the (easy) details.

Very recently, using modifications of the techniques of this paper, parallel algorithms with a linear time–processor product were obtained for the "treewidth $\leq k$" problem [11].

Acknowledgments. The presentation of these results was improved considably by comments from Jens Gustedt, Torben Hagerup, Ton Kloks, Dieter Kratsch, Bruce Reed, and an anonymous referee.

REFERENCES

[1] K. R. Abrahamson and M. R. Fellows, *Finite automata, bounded treewidth and well-quasiordering*, in Proc. AMS Summer Workshop on Graph Minors, Graph Structure Theory, Contemporary Mathematics, Vol. 147, American Mathematical Society, Providence, RI, 1993, pp. 539–564.

[2] S. Arnborg, *Efficient algorithms for combinatorial problems on graphs with bounded decomposability: A survey*, BIT, 25 (1985), pp. 2–23.

[3] S. Arnborg, D. G. Corneil, and A. Proskurowski, *Complexity of finding embeddings in a k-tree*, SIAM J. Algebraic Discrete Meth., 8 (1987), pp. 277–284.

[4] S. Arnborg, B. Courcelle, A. Proskurowski, and D. Seese, *An algebraic theory of graph reduction*, J. Assoc. Comput. Mach., 40 (1993), pp. 1134–1164.

[5] S. Arnborg, J. Lagergren, and D. Seese, *Easy problems for tree-decomposable graphs*, J. Algorithms, 12 (1991), pp. 308–340.

[6] S. Arnborg and A. Proskurowski, *Linear time algorithms for NP-hard problems restricted to partial k-trees*, Discrete Appl. Math., 23 (1989), pp. 11–24.

[7] H. L. Bodlaender, *Dynamic programming algorithms on graphs with bounded tree-width*, in Proc. 15th International Colloquium on Automata, Languages, and Programming, Lecture Notes in Comput. Sci. 317, Springer-Verlag, Berlin, 1988, pp. 105–119.

[8] ———, *A tourist guide through treewidth*, Acta Cybernetica, 11 (1993), pp. 1–23.

[9] ———, *Improved self-reduction algorithms for graphs with bounded treewidth*, Discrete Appl. Math., 54 (1994), pp. 101–115.

[10] H. L. Bodlaender, M. R. Fellows, and M. Hallett, *Beyond NP-completeness for problems of bounded width: Hardness for the W hierarchy*, in Proc. 26th Annual Symposium on Theory of Computing, IEEE Computer Society Press, Los Alamitos, CA, New York, 1994, pp. 449–458.

[11] H. L. Bodlaender and T. Hagerup, *Parallel algorithms with optimal speedup for bounded treewidth*, in Proc. 22nd International Colloquium on Automata, Languages, and Programming, Z. Fülöp and F. Gécseg, eds., Lecture Notes in Comput. Sci. 944, Springer-Verlag, Berlin, 1995, pp. 268–279; SIAM J. Comput., to appear.

[12] H. L. Bodlaender and T. Kloks, *Efficient and constructive algorithms for the pathwidth and treewidth of graphs*, J. Algorithms, 1996, to appear.

[13] H. L. Bodlaender and R. H. Möhring, *The pathwidth and treewidth of cographs*, SIAM J. Discrete Math., 6 (1993), pp. 181–188.

[14] 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. 555–581.

[15] B. Courcelle, *The monadic second-order logic of graphs* I*: Recognizable sets of finite graphs*, Inform. and Comput., 85 (1990), pp. 12–75.

[16] B. Courcelle and M. Mosbah, *Monadic second-order evaluations on tree-decomposable graphs*, Theoret. Comput. Sci., 109 (1993), pp. 49–82.

[17] M. R. Fellows and M. A. Langston, *Nonconstructive advances in polynomial-time complexity*, Inform. Process. Lett., 26 (1987), pp. 157–162.

[18] ———, *Nonconstructive tools for proving polynomial-time decidability*, J. Assoc. Comput. Mach., 35 (1988), pp. 727–739.

[19] ———, *An analogue of the Myhill–Nerode theorem and its use in computing finite-basis characterizations*, in Proc. 30th Annual Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1989, pp. 520–525.

[20] ———, *On search, decision and the efficiency of polynomial-time algorithms*, J. Comput. System Sci., 49 (1994), pp. 769–779.

[21] D. S. Johnson, *The NP-completeness column: An ongoing guide*, J. Algorithms, 8 (1987), pp. 285–303.

[22] T. Kloks, *Treewidth: Computations and Approximations*, Lecture Notes in Comput. Sci., 842, Springer-Verlag, Berlin, 1994.

[23] J. Lagergren, *Efficient parallel algorithms for graphs of bounded tree-width*, J. Algorithms, 20 (1996), pp. 20–44.

[24] J. Lagergren and S. Arnborg, *Finding minimal forbidden minors using a finite congruence*, in Proc. 18th International Colloquium on Automata, Languages, and Programming, Lecture Notes in Comput. Sci. 510, Springer-Verlag, Berlin, 1991, pp. 532–543.

[25] J. Matoušek and R. Thomas, *Algorithms finding tree-decompositions of graphs*, J. Algorithms, 12 (1991), pp. 1–22.

[26] B. Reed, *Finding approximate separators and computing tree-width quickly*, in Proc. 24th

Annual Symposium on Theory of Computing, Association for Computing Machinery, New York, 1992, pp. 221–228.

[27] N. Robertson and P. D. Seymour, *Graph minors* II: *Algorithmic aspects of tree-width*, J. Algorithms, 7 (1986), pp. 309–322.

[28] ———, *Graph minors* V: *Excluding a planar graph*, J. Combin. Theory Ser. B, 41 (1986), pp. 92–114.

[29] ———, *Graph minors* IV: *Tree-width and well-quasi-ordering*, J. Combin. Theory Ser. B, 48 (1990), pp. 227–254.

[30] ———, *Graph minors* X: *Obstructions to tree-decomposition*, J. Combin. Theory Ser. B, 52 (1991), pp. 153–190.

[31] ———, *Graph minors* XIII: *The disjoint paths problem*, J. Combin. Theory Ser. B, 63 (1995), pp. 65–110.

[32] D. P. Sanders, *On linear recognition of tree-width at most four*, SIAM J. Discrete Math., 9 (1996), pp. 101–117.

[33] P. Scheffler, *Die Baumweite von Graphen als ein Maß für die Kompliziertheit algorithmischer Probleme*, Ph.D. thesis, Akademie der Wissenschaften der DDR, Berlin, 1989.

[34] J. van Leeuwen, *Graph algorithms*, in Handbook of Theoretical Computer Science A: Algorithms and Complexity Theory, North–Holland, Amsterdam, 1990, pp. 527–631.

[35] T. V. Wimer, *Linear Algorithms on k-Terminal Graphs*, Ph.D. thesis, Department of Computer Science, Clemson University, Clemson, SC, 1987.

SIAM J. COMPUT.
Vol. 25, No. 6, pp. 1318–1331, December 1996

AN OPTIMAL $O(\log\log N)$-TIME PARALLEL ALGORITHM FOR DETECTING ALL SQUARES IN A STRING*

ALBERTO APOSTOLICO† AND DANY BRESLAUER‡

Abstract. An optimal $O(\log\log n)$-time concurrent-read concurrent-write parallel algorithm for detecting all squares in a string is presented. A tight lower bound shows that over general alphabets, this is the fastest possible optimal algorithm. When p processors are available, the bounds become $\Theta(\lceil (n\log n)/p\rceil + \log\log_{\lceil 1+p/n\rceil} 2p)$. The algorithm uses an optimal parallel string-matching algorithm together with periodicity properties to locate the squares within the input string.

Key words. squares, repititions, string matching, parallel algorithms, lower bounds

AMS subject classifications. 68Q10, 68Q20, 68Q25

1. Introduction.

A nonempty string of the form xx is called a *repetition.* Some strings, such as $a^n = aaa\ldots aa$, contain $\Omega(n^2)$ repetitions since they have $\Omega(n)$ repetitions starting at most positions. A *square* is defined as a repetition xx where x is primitive.[1] Strings that do not contain any repetition are called *repetition-free* or *square-free.* For example, "*aa*," "*abab*," and "*baba*" are squares which are contained in the string "*baababa*."

It is trivial to show that any string whose length is larger than three over alphabets of two symbols contains a square. However, there exist strings of infinite length on three-letter alphabets that are square-free, as shown by Thue [29, 30] at the beginning of the century. Since then, numerous works have been published on the subject and repetitions in strings have been found to be relevant to several fields, including coding theory, formal language theory, data compression, and combinatorics [1, 6, 7, 14, 15, 22, 23, 28].

The alphabet that the input symbols are chosen from has an important role in the design of efficient string algorithms. The literature distinguishes between four types of alphabets: *constant-size alphabets* that have a bounded number of symbols; *fixed alphabets*, where the symbols are assumed to be integers from a restricted range; *ordered alphabets*, where the alphabet is (arbitrarily) totally ordered and the only access an algorithm has to the input symbols is by order comparisons; and *general alphabets*, where the only access an algorithm has to the input symbols is by equality comparisons.

* Received by the editors December 21, 1993; accepted for publication (in revised form) March 17, 1995. The results presented in this paper were first reported at the 19th International Colloquium on Automata, Languages, and Programming.

† Department of Computer Science, Purdue University, West Lafayette, IN 47907 and Dipartimento di Elettronica e Informatica, Università di Padova, 35100 Padova, Italy. The research of this author was partially supported by NSF grants CCR-89-00305 and CCR-92-01078, NATO grant CRG 900293, the National Research Council of Italy, and EC ESPRIT III Basic Research Program contract 9072 (project GEPPCOM).

‡ Basic Research in Computer Science (BRICS), Centre of the Danish National Research Foundation, Department of Computer Science, University of Aarhus, DK-8000 Aarhus C, Denmark. The research of this author was partially supported by EC ESPRIT Basic Research Action Program contract 7141 (project ALCOM II). Parts of the research reported in this paper were carried out while the author was supported by the IBM Graduate Fellowship while studying at Columbia University, New York, NY, while visiting at the Istituto di Elaborazione dell'Informazione, Consiglio Nazionale delle Ricerche, Pisa, Italy with the support of the European Research Consortium for Informatics and Mathematics postdoctoral fellowship, and while visiting at the Università del'Aquila, L'Aquila, Italy.

[1] A string x is primitive if $x = u^k$ for some integer k implies that $k = 1$ and $x = u$.

In the last decade, several sequential algorithms that find all squares in strings have been published. Algorithms that were discovered by Apostolico and Preparata [4] and by Crochemore [13, 15] find all squares in a string of length n over ordered alphabets in $O(n \log n)$ time. Rabin [27] gave a randomized algorithm that takes $O(n \log n)$ expected time over constant-size alphabets. Any sequential algorithm that lists all squares in a string of length n must take at least $\Omega(n \log n)$ time since there exist strings, such as the *Fibonacci strings* [13], that contain $\Omega(n \log n)$ distinct squares.

Main and Lorentz [25] discovered an algorithm that finds all squares in strings over general alphabets in $O(n \log n)$ time. They also proved that over general alphabets $\Omega(n \log n)$ comparisons are necessary even to decide if a string is square-free. In another paper, Main and Lorentz [26] show that the problem of deciding whether a string is square-free can be solved in $O(n)$ time over constant-size alphabets. Crochemore [15] also gave a linear-time algorithm for the latter problem.

In parallel, algorithms by Crochemore and Rytter [16, 17] test if strings over ordered alphabets are square-free in $O(\log n)$ time using n processors. These algorithms use $O(n^{1+\epsilon})$ space. Apostolico [2] designed an algorithm that tests if a string is square-free and also detects all squares within the same time and processor bounds using linear auxiliary space. Apostolico's algorithm [2] assumes that the alphabet is ordered, a restriction that is not necessary to solve this problem. Apostolico's algorithm for testing if a string is square-free is more efficient over constant-size alphabets and achieves the $O(\log n)$ time bound using only $n/\log n$ processors. All of these parallel algorithms are designed for the concurrent-read concurrent-write parallel random-access machine (CRCW PRAM) computation model.

A parallel algorithm is said to be *optimal*, or to achieve an *optimal speedup*, if its time–processor product, which is the total number of operations performed, is equal to the running time of the fastest sequential algorithm for the same problem. All of the parallel algorithms that are mentioned above achieve an optimal speedup. Notice that squares can be trivially detected in constant time using a polynomial number of processors; our goal is to develop parallel algorithms that are efficient with respect to both time and processor complexities.

In this paper, we develop an optimal parallel algorithm that finds all squares in a string in $O(\log \log n)$ time. The new algorithm not only improves on the previous best bound of $O(\log n)$ time, but it is also the first efficient parallel algorithm for this problem over general alphabets. We derive a lower bound that shows that over general alphabets, this is the fastest possible optimal algorithm by a reduction to a lower bound that was given by Breslauer and Galil [11] for the string-matching problem. If p processors are available, then the bounds become

$$\Theta\left(\left\lceil \frac{n \log n}{p} \right\rceil + \log \log_{\lceil 1+p/n \rceil} 2p\right).$$

The paper is organized as follows. Section 2 overviews some known parallel algorithms and tools that are used by the new algorithm. Section 3 presents a simple version of the algorithm that tests if a string is square-free and §4 develops a more complicated version that finds all the squares. Section 5 is devoted to the lower bound and §6 gives tight bounds for any given number of processors. Concluding remarks are given in §7.

2. The CRCW PRAM model.

The algorithms described in this paper are for the CRCW PRAM model. We use the weakest version of this model, called the *common CRCW PRAM.* In this model, many processors have access to a shared memory. Concurrent-read and -write operations are allowed at all memory locations. If a few processors attempt to write simultaneously to the same memory location, then they all write the same value.

The square-detection algorithm uses a string-matching algorithm. The input to the string-matching algorithm consists of two strings, $pattern[1..m]$ and $text[1..n]$, and the output is a Boolean array $match[1..n]$ that has a "*true*" value at each position where an occurrence of the pattern starts in the text. We use Breslauer and Galil's [10] parallel string-matching algorithm, which takes $O(\log\log n)$ time using an $n/\log\log n$-processor CRCW PRAM. This algorithm is the fastest optimal parallel string-matching algorithm on general alphabets as shown by Breslauer and Galil [11]. If p processors are available, then the time bounds for the string-matching problem are $\Theta(\lceil n/p\rceil + \log\log_{\lceil 1+p/n\rceil} 2p)$.

The square-detection algorithm also uses an algorithm of Fich, Ragde, and Wigderson [19] to compute the minima of n integers in the range $1, \ldots, n$ in constant time using an n-processor CRCW PRAM. This minima algorithm, for example, can find the first occurrence of a string in another string: after the occurrences are computed by the string-matching algorithm mentioned above, look for the smallest i such that $match[i] =$ "*true*."

Finally, we use the following theorem.

THEOREM 2.1 (Brent [8]). *Any parallel algorithm of time t that consists of a total of x elementary operations can be implemented on p processors in $\lceil x/p\rceil + t$ time.*

If we return to the example above, which finds the first occurrence of one string in another, we see that the second step of finding the smallest index of an occurrence takes constant time using n processors, while the use of the string-matching procedure takes $O(\log\log n)$ time using $n/\log\log n$ processors. By Theorem 2.1, the second step can be slowed down to work in $O(\log\log n)$ time using $n/\log\log n$ processors.

3. Testing if a string is square-free.

This section describes an algorithm that tests if a string $S[1..n]$ is square-free. The algorithm that finds all squares is more involved and is given in §4.

THEOREM 3.1. *There exists an algorithm that tests if a string $S[1..n]$ over a general alphabet is square-free in $O(\log\log n)$ time using $n\log n/\log\log n$ processors.*

Proof. The algorithm consists of independent stages which are computed simultaneously. In stage number η, $0 \le \eta \le \lceil \log_2 n\rceil - 1$, the algorithm looks only for repetitions xx such that $2l_\eta - 1 \le |x| < 2l_{\eta+1} - 1$ and $l_\eta = 2^\eta$. If some repetition is found, then a global variable is set to indicate that the string is not square-free. Notice that the complete range of possible lengths of x is covered, and if there exists a repetition, it will be discovered.

We show how to implement stage number η in $T_\eta = O(\log\log l_\eta)$ time and $O(n)$ operations. Since there are $O(\log n)$ stages, the total number of operations is $O(n\log n)$. By Theorem 2.1, the algorithm can be implemented in $\max T_\eta = O(\log\log n)$ time using $n\log n/\log\log n$ processors. □

3.1. The stages.

We describe stage number η, $0 \le \eta \le \lceil \log_2 n\rceil - 1$, which looks only for repetitions xx such that $2l_\eta - 1 \le |x| < 2l_{\eta+1} - 1$. To simplify the presentation, assume without loss of generality that the algorithm can access symbols whose indices are out of the boundaries of the input string. Comparisons with such symbols are answered as unequal.

Partition the input string $S[1..n]$ into consecutive blocks of length l_η. That is, block number k for $1 \le k < \lfloor n/l_\eta\rfloor$ is $S[(k-1)l_\eta + 1..kl_\eta]$. Let $B = S[P..P + l_\eta - 1]$ be one of these blocks. A repetition xx is said to be *hinged* on B if $2l_\eta - 1 \le |x| < 2l_{\eta+1} - 1$

and B is fully contained in the first copy of x. Stage number η consists of substages which are also computed simultaneously. There is a substage for each block of length l_η. Each substage checks if there is any repetition which is hinged on the block that it is assigned to.

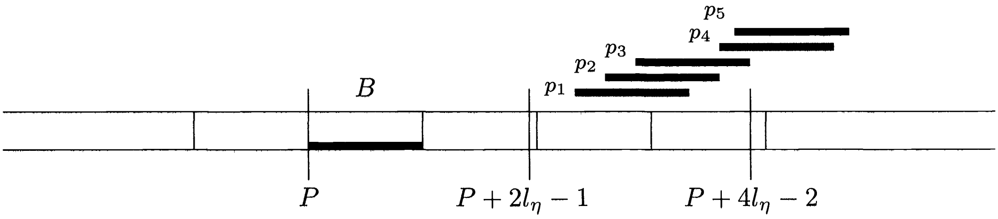


FIG. 1. *The substage which is assigned to the block* $B = S[P..P + l_\eta - 1]$ *finds all occurrences of* B *that start between positions* $P + 2l_\eta - 1$ *and* $P + 4l_\eta - 2$.

The substage which is assigned to block B starts with a call to the string-matching algorithm to find all occurrences of B in $S[P + 2l_\eta - 1..P + 5l_\eta - 3]$. Let $p_1 < p_2 < \cdots < p_r$ be the indices of these occurrences. Then $P + 2l_\eta - 1 \leq p_i < P + 4l_\eta - 1$, for $i = 1, \ldots, r$. See Figure 1.

Notice that for each repetition xx that is hinged on B, there must be an occurrence of B at position $P + |x|$. This occurrence is included in the $\{p_i\}$ sequence.

LEMMA 3.2. *For each* p_i, *we can test in constant time and* $O(l_\eta)$ *operations if there is any repetition* xx *that is hinged on* B *such that* $|x| = p_i - P$.

Proof. Let $l = p_i - P$. We are looking for repetitions xx such that $|x| = l$. For all ζ's in the range $P + l_\eta - 1 \leq \zeta \leq p_i$, check if $S[\zeta - l] = S[\zeta]$ and if $S[\zeta] = S[\zeta + l]$. Let ζ_L be the largest index in this range such that $S[P + l_\eta..\zeta_L] = S[P + l_\eta + l..\zeta_L + l]$ and ζ_R be the smallest index such that $S[\zeta_R..p_i - 1] = S[\zeta_R - l..P - 1]$. We can find ζ_L and ζ_R in constant time and $O(l_\eta)$ operations using the integer-minima algorithm of Fich, Ragde, and Wigderson [19].

We show that there are repetitions xx that are hinged on B such that $|x| = l$ if and only if $\zeta_R \leq \zeta_L + 1$. Moreover, these repetitions start at positions s for $\zeta_R - l \leq s \leq \zeta_L - l + 1$.

If there is a repetition xx that is hinged on B starting at position s such that $|x| = l$, then $S[\zeta - l] = S[\zeta]$ for all ζ's in the range $s + l \leq \zeta < p_i$ and $S[\zeta] = S[\zeta + l]$ for all ζ's in the range $P + l_\eta \leq \zeta < s + l$. Then, however, $\zeta_L \geq s + l - 1$ and $\zeta_R \leq s + l$, and thus $\zeta_R - l \leq s \leq \zeta_L - l + 1$ and $\zeta_R \leq \zeta_L + 1$. See Figure 2.

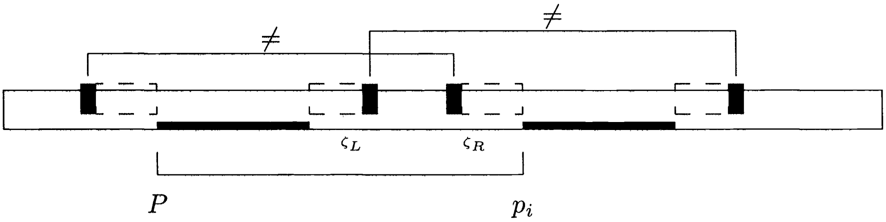


FIG. 2. *If* $\zeta_R > \zeta_L + 1$, *then there is no repetition* xx *that is hinged on the block* B *such that* $|x| = p_i - P$.

On the other hand, if $\zeta_R \leq \zeta_L + 1$, then $S[\zeta_R - l..\zeta_L] = S[\zeta_R..\zeta_L + l]$. (Recall that there is an occurrence of $S[P..P + l_\eta - 1]$ at position p_i and thus $S[P..P + l_\eta - 1] = S[p_i..p_i + l_\eta - 1]$.) The last equality means that there are repetitions xx such that $|x| = l$, starting at positions s for $\zeta_R - l \leq s \leq \zeta_L - l + 1$. □

The algorithm can check if any of the p_i's corresponds to a repetition in constant time using Lemma 3.2, but it would make $O(rl_\eta)$ operations if the length of the $\{p_i\}$ sequence is r. Luckily, for now, the algorithm has only to test if the string is square-free and it does not have to check if all the p_i's correspond to repetitions; if $r > 2$, then $S[1..n]$ must contain a square, as the following lemma shows.

LEMMA 3.3. *If the length of the $\{p_i\}$ sequences is $r > 2$, then $S[1..n]$ contains a repetition. This repetition is shorter than the repetitions that are supposed to be found in this stage.*

Proof. Recall that $P + 2l_\eta - 1 \leq p_i < P + 4l_\eta - 1$ for $i = 1, \ldots, r$. If $r \geq 3$, then either $p_2 - p_1 \leq l_\eta$ or $p_3 - p_2 \leq l_\eta$. Then, however, there is a repetition xx such that $|x| = p_2 - p_1$ or $|x| = p_3 - p_2$ (respectively), starting at position p_1 or p_2 (respectively). □

The computation in each substage of stage η can be summarized as follows:

1. Compute the $\{p_i\}$ sequence.
2. If the $\{p_i\}$ sequence has more than two elements, then by Lemma 3.3, the string $S[1..n]$ contains a repetition. This repetition will also be found by some stage number μ, $\mu < \eta$.
3. If the $\{p_i\}$ sequence has at most two elements, check if these elements correspond to repetitions using the procedure described in Lemma 3.2.

LEMMA 3.4. *Stage number η is correct. It takes $O(\log\log l_\eta)$ time and makes $O(n)$ operations.*

Proof. For correctness, we have to show that if the string $S[1..n]$ contains any repetition xx such that $2l_\eta - 1 \leq |x| < 2l_{\eta+1} - 1$, then some repetition will be found. Assume that there is such a repetition. Since $2l_\eta - 1 \leq |x|$, there must be a block of length l_η that is completely contained in the first x. The substage which is assigned to that block will either find the repetition xx or conclude that there is a shorter repetition by Lemma 3.3. In both cases, some repetition has been found. Notice that some repetitions can be detected by several stages and substages simultaneously.

Stage number η consists of $\lfloor n/l_\eta \rfloor$ independent substages. In each substage, step number 1 takes $O(\log\log l_\eta)$ time and $O(l_\eta)$ operation using Breslauer and Galil's string-matching algorithm. Steps number 2 and 3 take constant time and make $O(l_\eta)$ operations. Since all of the substages are computed in parallel, stage number η takes $O(\log\log l_\eta)$ time and makes $O(n)$ operations. □

4. Detecting all squares. In this section, we show how the algorithm that was given in §3 can be generalized to find all squares in a string.

Beame and Håstad [5] proved a lower bound of $\Omega(\log n/\log\log n)$ time for computing the parity of n input bits on CRCW PRAMs with any polynomial number of processors. This lower bound implies that many "interesting" problems would require at least that time. However, several string problems, including the problem of detecting all squares in a string, have constant-time solutions using a polynomial number of processors.

While the problem of testing if a string is square-free has only a single output bit, the problem of finding all squares has a more complicated output structure. If we wish to obtain algorithms that get around Beame and Håstad's lower bound, we cannot count the number of squares that are found and therefore we can not list them contiguously in an array. Instead, we will represent the output of the algorithm in a sparse array with $O(n \log n)$ entries. Notice that this problem did not exist in the previous square-detection algorithms since their time bounds were at least $O(\log n)$.

Similarly to the testing algorithm, the square-detection algorithm proceeds in independent stages which are computed within the same time and processor bounds as before, only now, since the algorithm must find all the squares, the following difficulties arise.

1. The detection algorithm cannot use Lemma 3.3 only to conclude that the string is not square-free; it must find all the squares.
2. The algorithm has to verify which repetitions are squares. This was not necessary before since a string is square-free if and only if it is repetition-free.
3. The squares have to be represented in a sparse array with $O(n \log n)$ entries.

The first two issues will be addressed in §4.1, which describes the stages of the square-detection algorithm, while the third issue is discussed next.

The following lemma is used to justify the output representation used by the algorithm.

LEMMA 4.1 (see, e.g., Crochemore and Rytter [18]). *If there are three squares xx, yy, and zz such that $|x| < |y| < |z|$ that start at the same position of some string, then $|x| + |y| \leq |z|$.*

Recall that in stage number η, the algorithm looks only for squares xx such that $2l_\eta - 1 \leq |x| < 2l_{\eta+1} - 1$ and $l_\eta = 2^\eta$. Therefore, by Lemma 4.1, there are no more than two squares that start at each position of the input string and have to be discovered in the same stage. Thus the output can be represented in an array that will for each position of the input string and for each stage hold the two squares that might be detected starting at the specific position in the specific stage. (For example, let u be primitive and v be a nonempty proper prefix of u. Then the string $u^k v u^{k+1} v u$, $k \geq 1$, contains the two prefix squares $u^k v u^k v$ and $u^k v u u^k v u$ whose lengths differ by $2|u|$. If $k \geq 2$, then it contains also the prefix square uu, and if $k = 2$, then the inequality in Lemma 4.1 is tight. In the extreme case, by letting $u =$ "ab" and $v =$ "a," we get arbitrary long pairs of squares whose lengths differ by 4.)

The complexity bounds of the square-detection algorithm are summarized in the following theorem.

THEOREM 4.2. *There exists an algorithm that finds all squares in a string $S[1..n]$ over a general alphabet in $O(\log \log n)$ time using $n \log n / \log \log n$ processors.*

4.1. The stages. Consider a single stage. As in §3.1, the input string $S[1..n]$ is partitioned into consecutive blocks of length l_η and there is a substage that is assigned to each such block. To simplify the presentation, we allow squares to be discovered by several substages simultaneously: the substage that is assigned to block B discovers all the squares which are hinged on this block. Later, we make sure that the information about each square is written only once into the output array by reporting only those squares for which B is the leftmost block fully contained in the square. Thus stage number η finds all squares xx such that $2l_\eta - 1 \leq |x| < 2l_{\eta+1} - 1$.

As already noted, each square that is hinged on B ties block B to a specific replica. The substage that is assigned to B starts with a call to the string-matching algorithm to find the viable replicas of B. Let $p_1 < \cdots < p_r$ denote their indices.

DEFINITION 4.3. *A string x is a* rotation *of another string $\hat{x}$ (and vice versa) if $x = uv$ and $\hat{x} = vu$ for some strings u and v.*

DEFINITION 4.4. *A string S has* a period *u if S is a prefix of u^k for some large enough k. Alternatively, a string $S[1..n]$ has a period of length π if $S[i] = S[i + \pi]$ for $i = 1, \ldots, n - \pi$. The shortest period of a string S is called* the period *of S.*

LEMMA 4.5 (Lyndon and Schützenberger [24]). *If a string of length m has two periods of lengths p and q and $p + q \leq m$, then it also has a period of length $gcd(p, q)$.*

The task of the substage is to identify which of the p_i's corresponds to squares that are hinged on B. In Lemma 3.2, we have shown that it is possible to efficiently verify that some specific p_i corresponds to repetitions xx that are hinged on B such that $|x| = p_i - P$. The proof of Lemma 3.2 reveals that those differences $p_i - P$ that pass the repetition-detection test actually expose an entire sequence of repetitions which are consecutive rotations of the same repetition. Such a sequence will be called a *family* of repetitions.

LEMMA 4.6. *A family of repetitions contains a square if and only if all the repetitions in the family are squares.*

Proof. Let xx be a repetition but not a square. Thus $x = z^l$ and $l > 1$. If $\hat{x}$ is a rotation of x, then $\hat{x} = v(uv)^j(uv)^{l-j-1}u = (vu)^l$, where $z = uv$, and thus $\hat{x}$ is not primitive. □

The last lemma means that if we wish to certify that repetitions are actually squares, it is enough to certify one repetition in each family. The next lemma shows how to efficiently test that a given repetition is indeed a square by solving a single string-matching problem. (The technique for primitive certification proposed by Apostolico [2] uses information about shorter squares which are discovered in other stages. We use a different method that keeps the stages in the algorithm completely independent.)

LEMMA 4.7. *Given a repetition xx, let l be the index of the first occurrence of x in xx other than the trivial occurrence at the beginning of xx. Then, xx is a square if and only if $l = |x|$.*

Proof. Clearly, $l \leq |x|$. If $x = z^j$, then $xx = z^{2j}$ and x occurs at position $|z|$ of xx. On the other hand, if $l < |x|$, then xx has periods of lengths l and $|x|$ and by Lemma 4.5, l divides $|x|$. Then, however, $x = z^{|x|/l}$ is not primitive. □

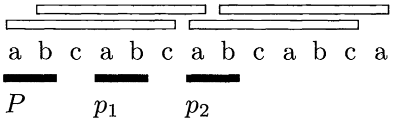


FIG. 3. *Repetitions must be certified to be squares. In this example, the repetitions in the family that corresponds to $p_2 - P$ are not squares.*

Given a replica of B at position p_i, we can find the family of repetitions xx such that $|x| = p_i - P$ using Lemma 3.2, and then we can certify that these repetitions are actually squares using Lemma 4.7. See Figure 3.

However, if the length of the $\{p_i\}$ sequence is large, then repeating the process above for each p_i can be costly. Moreover, it is a problem even to find and manipulate the $\{p_i\}$ sequence efficiently. The following lemmas will help to overcome this difficulty.

LEMMA 4.8. *Assume that the period length of a string $W[1..l]$ is p. If $W[1..l]$ occurs only at positions $p_1 < p_2 < \cdots < p_k$ of a string V and $p_k - p_1 \leq \lceil l/2 \rceil$, then the p_i's form an arithmetic progression with difference p.*

Proof. Assume $k \geq 2$. We prove that $p = p_{i+1} - p_i$ for $i = 1, \ldots, k-1$. The string W has periods of lengths p and $q = p_{i+1} - p_i$. Since $p \leq q \leq \lceil l/2 \rceil$, by Lemma 4.5, it also has a period of length $gcd(p, q)$. However, p is the length of the shortest period, so $p = gcd(p, q)$ and p must divide q. The string $V[p_i..p_{i+1} + l - 1]$ has period of length p. If $q > p$, then there must be another occurrence of W at position $p_i + p$ of V—a contradiction. □

Recall that $P + 2l_\eta - 1 \leq p_i < P + 4l_\eta - 1$. To utilize the last lemma, it is convenient to partition the sequence $\{p_i\}$ and to regard the substage as consisting of four consecutive *phases*. Each phase handles viable replicas of B in a subblock of size $l_\eta/2$ (hereafter, an $l_\eta/2$-block). We describe a generic phase involving the occurrences of B at positions $q_1 < \cdots < q_k$, where $\{q_i\}$ is a subsequence of $\{p_i\}$ that lists all the occurrences that fall within a $l_\eta/2$-block. (In the first stages, there are fewer phases.)

LEMMA 4.9. *The sequence $\{q_i\}$ of occurrences of B in an $l_\eta/2$-block is an arithmetic progression with difference q, where q is the period length of B.*

Proof. Lemma 4.9 is an immediate consequence of Lemma 4.8. □

The sequence $\{q_i\}$ can be represented using three integers: the start, the difference, and the sequence length. This representation can be easily computed from the output of the string-matching algorithm (which is a Boolean vector) using Fich, Ragde, and Wigderson's integer-minima algorithm [19] in constant time using $O(l_\eta)$ operations. This idea has also been successfully applied in efficient parallel algorithms for other string problems [3, 9, 12].

If the $\{q_i\}$ sequence does not contain any elements, then the phase does not need to do anything. If there is one element q_1, then the algorithm finds the family of repetitions that are associated with the difference $q_1 - P$ and certifies them to be squares as described above. The next lemmas are used in phases that have longer $\{q_i\}$ sequences.

Assume that the length of the arithmetic progression $\{q_i\}$ is $k \geq 2$ and let q be the difference of the progression. By Lemmas 4.8 and 4.9, the block $B = S[P..P + l_\eta - 1]$ and the substring covered by the occurrences of this block at positions q_i, $S[q_1..q_k + l_\eta - 1]$, have period length q. The algorithm proceeds by checking how far this periodicity extends on both sides of these substrings.

Let α_L and α_R be the positions where the periodicity of length q terminates on the left and on the right of B, respectively, and let γ_L and γ_R be the positions where the periodicity of length q terminates on the left and on the right of the substring $S[q_1..q_k + l_\eta - 1]$, respectively. We are interested in these indices only if $P - (q_k - P) + l_\eta \leq \alpha_L$, $\alpha_R < q_1 + l_\eta$, $P \leq \gamma_L$, and $\gamma_R < 2q_k - P$, and these indices are undefined otherwise. Namely, if all indices are defined, then $S[\alpha_L + 1..\alpha_R - 1]$ has period length q, $S[\alpha_L] \neq S[\alpha_L + q]$, $S[\alpha_R] \neq S[\alpha_R - q]$,

$$P - (q_k - P) + l_\eta \leq \alpha_L < P \quad \text{and} \quad P + l_\eta \leq \alpha_R < q_1 + l_\eta,$$

$S[\gamma_L + 1..\gamma_R - 1]$ has period length q, $S[\gamma_L] \neq S[\gamma_L + q]$, $S[\gamma_R] \neq S[\gamma_R - q]$, and

$$P \leq \gamma_L < q_1 \quad \text{and} \quad q_k + l_\eta \leq \gamma_R < 2q_k - P.$$

It is possible to compute the indices α_L, α_R, γ_L, and γ_R or to decide which indices are undefined in constant time and $O(l_\eta)$ operations using Fich, Ragde, and Wigderson's integer-minima algorithm [19].

The following lemmas classify the possible interactions between α_L, α_R, γ_L, and γ_R and their effect on the squares that are hinged on B.

LEMMA 4.10. *If one of α_R and γ_L is defined, then so is the other one, and $\alpha_R - \gamma_L \leq q$.*

Proof. By the definition of α_R and γ_L, $S[P..\alpha_R - 1]$ and $S[\gamma_L + 1..q_k + l_\eta - 1]$ have period length q, $S[\alpha_R] \neq S[\alpha_R - q]$ and $S[\gamma_L] \neq S[\gamma_L + q]$.

If $q < \alpha_R - \gamma_L$, then by the periodicity of $S[P..\alpha_R - 1]$ and since $\gamma_L + q < \alpha_R$, we get that $S[\gamma_L] = S[\gamma_L + q]$, in contradiction to the definition of α_R and γ_L. Therefore, $\alpha_R - \gamma_L \leq q$ or at least one of α_R and γ_L is undefined.

If α_R is undefined, then $S[P..q_1 + l_\eta - 1]$ has period length q and the argument above shows that γ_L can not be defined. The proof of the symmetric case is identical. □

The following lemma identifies certain repetitions that can never be squares.

LEMMA 4.11. *If both α_R and γ_L are undefined, then none of the repetitions possibly hinged on B is a square.*

Proof. If α_R and γ_L are undefined, then $S[P..q_k + l_\eta - 1]$ has period length q. Consider any q_i and let $l = q_i - P$. By the periodicity above and since $S[P..P+l_\eta-1] = S[q_i..q_i + l_\eta - 1]$, we get that $S[P..P + q_k - q_i + l_\eta - 1] = S[q_i..q_k + l_\eta - 1]$. Thus the substring $S[P..q_k + l_\eta - 1]$ has a period of length l. However, $q \le l_\eta/2 < l$ and by Lemma 4.5, q divides l.

Let xx be a repetition that is hinged on B starting at position s such that $|x| = l$. Then $x = S[s..s + l - 1] = S[s + l..P + l - 1]S[P..s + l - 1]$ has period length q and therefore x is not primitive. □

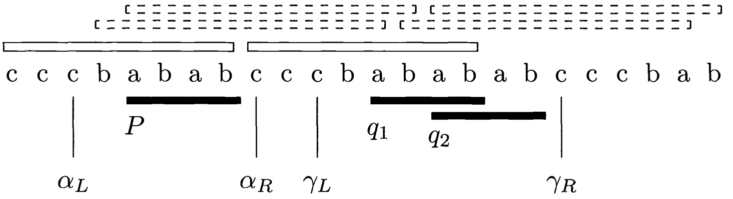


FIG. 4. *There can be at most two families of synchronized squares. In this example, one family corresponds to $\gamma_L - \alpha_L = q_1 - P$ and the other to $\gamma_R - \alpha_R = q_2 - P$.*

If both α_R and γ_L are defined, then certain repetitions, which are characterized in the next lemma, must align α_R with γ_R and α_L with γ_L. These repetitions are called *synchronized repetitions.* See Figure 4.

It is convenient to state the next lemmas in terms of the positions where the repetitions are centered; a repetition xx that starts at position s is *centered* at position $s + |x|$.

LEMMA 4.12. *If both α_R and γ_L are defined, then we have the following:*

1. *Repetitions that are hinged on B and centered at positions h such that $h \le \gamma_L$ may exist only if α_L is defined. These repetitions constitute a family of repetitions that corresponds to the difference $q_i - P$ provided that there exists some q_i such that $\gamma_L - \alpha_L = q_i - P$.*

2. *Repetitions that are hinged on B and centered at positions h such that $\alpha_R < h$ may exist only if γ_R is defined. These repetitions constitute a family of repetitions that corresponds to the difference $q_j - P$ provided that there exists some q_j such that $\gamma_R - \alpha_R = q_j - P$.*

Notice that if $\alpha_R < \gamma_L$, then repetitions whose center h satisfies $\alpha_R < h \le \gamma_L$ may exist only if both α_L and γ_R are defined and if $\gamma_R - \alpha_R = \gamma_L - \alpha_L$.

Proof. Let $xx = S[h-l..h+l-1]$ be a repetition that is hinged on B and centered at position h such that $|x| = q_i - P$, and let $l = |x|$.

Assume $P + l_\eta \le h \le \gamma_L$. The proof distinguishes between two cases. If α_L is undefined or if $\alpha_L < \gamma_L - l$ (see Figure 5), then by the periodicity in the definition of α_L and γ_L, $S[\gamma_L - l + q] = S[\gamma_L + q]$ and $S[\gamma_L - l] = S[\gamma_L - l + q]$. Since there is the repetition xx, we also have that $S[\gamma_L - l] = S[\gamma_L]$. Thus $S[\gamma_L] = S[\gamma_L + q]$, in contradiction to the fact that $S[\gamma_L] \ne S[\gamma_L + q]$ by the definition of γ_L.

Similarly, if $\alpha_L > \gamma_L - l$, then by the periodicity in the definition of α_L and γ_L, $S[\alpha_L + q] = S[\alpha_L + l + q]$ and $S[\alpha_L + l] = S[\alpha_L + l + q]$. Since there is the repetition

xx, we also have that $S[\alpha_L] = S[\alpha_L + l]$. Thus $S[\alpha_L] = S[\alpha_L + q]$, in contradiction to the fact that $S[\alpha_L] \neq S[\alpha_L + q]$ by the definition of α_L.

Therefore, such a repetition xx may exist only if $\alpha_L = \gamma_L - l$ or, in other words, if $\gamma_L - \alpha_L = q_i - P$ for some q_i. Since α_L, γ_L, and P are given, there is at most one such q_i.

The proof of the second part, where $\alpha_R < h$, is similar. □

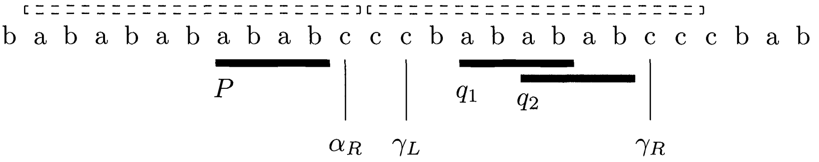


FIG. 5. *Illustrating one of the cases in Lemma* 4.12: *an undefined* α_L *is incompatible with any repetition hinged on* B *and centered at a position* $h \leq \gamma_L$.

As a consequence of the last lemma, there can be at most two repetition families (in each phase) that have to be verified and certified to be squares. However, there are squares which might have been missed since Lemma 4.12 did not cover all eventualities. If $\gamma_L < \alpha_R$, then there might exist repetitions whose center h satisfies $\gamma_L < h \leq \alpha_R$. These repetitions are called *unsynchronized repetitions.* We classify these repetitions next and show that if such repetitions exist, then they must be squares.

LEMMA 4.13. *If* α_R *and* γ_L *are defined and* $\gamma_L < \alpha_R$, *then there might be a family of repetitions associated with each of the differences* $l = q_i - P$ *with centers at positions* h *such that* $\gamma_L < h \leq \alpha_R$. *The repetitions in each such family are all squares, and they are centered at positions* h *such that* $\max(\alpha_L + l, \gamma_L) < h \leq \min(\alpha_R, \gamma_R - l)$. *Notice that such a family is not empty if and only if* $l < \min(\alpha_R - \alpha_L, \gamma_R - \gamma_L)$.

Proof. Consider repetitions $S[h - l..h - 1] = S[h..h + l - 1]$ that are associated with the difference $l = q_i - P$ and whose centers h satisfy $\gamma_L < h \leq \alpha_R$. We show that such repetitions exist if and only if $\alpha_L + l < h$ and $h \leq \gamma_R - l$ (ignoring the constraints involving undefined indices).

If $h \leq \alpha_L + l$, then $S[\alpha_L] = S[\alpha_L + l]$. Since $\gamma_L < h$, we know that $S[\alpha_L + l] = S[\alpha_L + l + q]$. Then, however, $S[\alpha_L] = S[\alpha_L + q]$, in contradiction to the definition of α_L. Similarly, it is impossible that $\gamma_R - l < h$.

On the other hand, if $\max(\alpha_L + l, \gamma_L) < h \leq \min(\alpha_R, \gamma_R - l)$, then $S[h - l..h - 1]$ and $S[h..h + l - 1]$ have period length q. Since $S[P..P + l_\eta - 1] = S[q_i..q_i + l_\eta - 1]$, we get that $S[h - l..h - 1] = S[h..h + l - 1]$. (The same reasoning also holds if α_L or γ_R are not defined.)

It remains to show that these repetitions are actually squares. If $S[h-l..h-1] = z^j$ for some $j > 1$, then $S[h - l..h - 1]$ has periods of length q and $|z|$ and by Lemma 4.5, q divides $|z|$. Then, however, $S[h - q..h - 1] = S[h..h + q - 1]$ and $\alpha_R - \gamma_L \geq 2q$, in contradiction to Lemma 4.10. □

The computation in each substage of the square-detection algorithm can be summarized as follows:

1. Compute the $\{p_i\}$ sequence and proceed in four phases.
2. In each phase, find the arithmetic progression $\{q_i\}$.
3. If the $\{q_i\}$ sequence has a single element q_1, then find the repetition family that corresponds to q_1 using Lemma 3.2 and certify that these repetitions are squares using Lemma 4.7.
4. If the $\{q_i\}$ sequence has at least two elements, then do the following:

(a) Find the synchronized repetition families using Lemma 4.12 and certify that these repetitions are squares using Lemma 4.7.

(b) Find the unsynchronized squares using Lemma 4.13.

LEMMA 4.14. *Stage number η is correct. It takes $O(\log\log l_\eta)$ time and makes $O(n)$ operations.*

Proof. It is clear that if the string $S[1..n]$ contains any square xx such that $2l_\eta - 1 \leq |x| < 2l_{\eta+1} - 1$, then there must be a block B of length l_η that is the leftmost block completely contained in the square. We have seen that the substage that is assigned to the block B will find xx.

Stage number η consists of $\lfloor n/l_\eta \rfloor$ independent substages. Each substage might make at most nine calls to Breslauer and Galil's string-matching algorithm: one to find the $\{p_i\}$ sequence and at most two in each phase to certify squares using Lemma 4.7. These calls take $O(\log\log l_\eta)$ time and make $O(l_\eta)$ operations. The rest of the work in each substage takes constant time and $O(l_\eta)$ operations. Since all of the substages are computed in parallel, stage number η takes $O(\log\log l_\eta)$ time and makes $O(n)$ operations. □

Remark. Assume that the sequence $\{q_i\}$ has $k > 1$ elements and difference q. If α_R and γ_L are defined, then some synchronizing repetitions might have to be certified to be squares. It easy to check that for the repetitions xx that arise in this case, if $x = z^j$, then $j \leq \epsilon$ for some small positive constant ϵ. Thus it is sufficient to verify that $x \neq z^j$ for $j = 2, \ldots, \epsilon$ in order to certify that x is primitive. This is more efficient than the general square-certification method suggested in Lemma 4.7.

5. The lower bound.

We prove a lower bound for testing if a string is square-free by a reduction to Breslauer and Galil's [11] lower bound for string matching. Breslauer and Galil show that an adversary can fool any algorithm which claims to check if a string has a period that is shorter than half of its length in fewer than $\Omega(\lceil n/p \rceil + \log\log_{\lceil 1+p/n \rceil} 2p)$ rounds with p comparisons in each round. The lower bound holds for the CRCW PRAM model in the case of general alphabets, where the only access an algorithm has to the input string is by pairwise symbol comparisons.

We will not report the details of that lower bound. We only use the fact that the adversary generates a string $S[1..n]$ that has the following property: if $S[i] = S[j]$, then $S[k] = S[i]$ for any integer k such that $k \equiv i \pmod{|j-i|}$ and $1 \leq k \leq n$.

LEMMA 5.1. *The string generated by Breslauer and Galil's adversary has a period that is shorter than half of its length if and only if it contains a square.*

Proof. If the string generated by the adversary has a period which is shorter than half of its length, then it contains a square that starts at the beginning of the string.

On the other hand, assume that a square xx starts at position s of $S[1..n]$. Namely, $S[s+k] = S[s+|x|+k]$ for $k = 0, \ldots, |x|-1$. Then, however, by the property mentioned above, the string generated by the adversary has a period of length $|x|$, which is smaller than half of the string length. □

We are now ready to prove the lower bound.

THEOREM 5.2. *Any parallel algorithm that tests if a string $S[1..n]$ over general alphabets is square-free must take $\Omega(\lceil (n\log n)/p \rceil + \log\log_{\lceil 1+p/n \rceil} 2p)$ rounds with p comparisons in each round.*

Proof. Main and Lorentz [26] show that any sequential algorithm that tests if a string over general alphabets is square-free must make $\Omega(n \log n)$ comparisons. This gives an immediate lower bound of $\Omega(\lceil (n\log n)/p \rceil)$ rounds with p comparisons in each round.

By Lemma 5.1, the string that is generated by the adversary of Breslauer and

Galil has a period that is shorter than half of its length if and only if it contains a square. Breslauer and Galil show that after $\Omega(\log\log_{\lceil 1+p/n\rceil} 2p)$ rounds, the adversary still has the choice of forcing the string to have a period that is shorter than half of its length or not to have any such period. Therefore, any algorithm that tries to decide in fewer rounds if a string is square-free can be fooled. By combining these two bounds, we get the claimed lower bound. □

COROLLARY 5.3. *Any optimal parallel algorithm that tests if a string $S[1..n]$ is square-free must take $\Omega(\log\log n)$ rounds.*

Proof. By Theorem 5.2, the lower bound is $\Omega(\log\log n)$ even with $n\log n$ comparisons in each round. □

6. The number of processors. This section derives tight bounds for any given number of available processors.

THEOREM 6.1. *If p processors are available, then the lower and upper bounds for testing if a string is square-free and for detecting all squares are*

$$\Theta\left(\left\lceil\frac{n\log n}{p}\right\rceil + \log\log_{\lceil 1+p/n\rceil} 2p\right).$$

Proof. The lower bound was given in Theorem 5.2. It remains to prove the upper bound.

1. If $p \le n\log n/\log\log n$, then by Theorem 2.1, the optimal algorithms of §§3 and 4 can be slowed down to run in $O((n\log n)/p)$ time, matching the lower bound.

2. If $n\log n/\log\log n < p \le n\log n$, then the lower bound is $\Omega(\log\log n)$, matching the time bound of the algorithms with only $n\log n/\log\log n$ processors.

If $p > n\log n$, then we must go back to the algorithms given in §§3 and 4. The processors are distributed equally among the stages. In stage number η, the processors are distributed equally among the substages, giving $(p/n\log n)l_\eta$ processors to each substage.

Since substages that handle strings of length $O(l_\eta)$ have more than l_η processors available, the substages take constant time except for the calls to Breslauer and Galil's string-matching algorithm. These calls take $T_\eta = O(\log\log_{\lceil 1+p/n\log n\rceil} 2(p/n\log n)l_\eta)$ time. Therefore, the whole algorithm takes $\max T_\eta = O(\log\log_{\lceil 1+p/n\log n\rceil}(2p/\log n))$ time.

3. If $p > n\log n$, then we can verify that $\log\log_{\lceil 1+p/n\log n\rceil}(2p/\log n) \in O(\log\log_{\lceil 1+p/n\rceil} 2p)$, establishing that the lower and upper bounds are the same.

4. If $p > n^{1+\epsilon}$ for some fixed $\epsilon > 0$, then the upper bound is $O(1)$. □

7. Concluding remarks. The algorithm described in this paper uses a string-matching procedure as a "black box" that has a specific input–output functionality without going into its implementation details. Breslauer and Galil's string-matching algorithm is the fastest possible over general alphabets; however, it is unknown at the moment if a faster algorithm exists over constant-size alphabets. If such an algorithm exists, it could be used in a faster algorithm for finding squares. Notice that a fast CRCW PRAM implementation requires the computation of certain functions such as the log function and integral powers within the time and processor bounds. Regardless of the feasibility of such a computation, the algorithm that was described in this paper is valid in the parallel comparison decision-tree model.

Our parallel square-detection algorithm resembles the sequential algorithms of Main and Lorentz [25, 26]. (The testing algorithm is, in fact, a parallel implementation of the testing algorithm in [26].) Still, the sequential implementation of our parallel algorithm is interesting on its own. By using a time–space-optimal string-matching algorithm, such as the algorithm of Galil and Seiferas [21], we obtain a time–space-optimal algorithm for detecting squares. By using a real-time string-matching algorithm, such as the algorithm of Galil [20], and a careful treatment of periods within the input string, we obtain an on-line square-detection algorithm that reports squares as soon as they are formed, while the input string is extended even on both sides, spending $O(\log n)$ time per symbol. No such algorithms were known before.

Acknowledgments. We thank Zvi Galil, Roberto Grossi, and Kunsoo Park for their valuable comments.

REFERENCES

[1] A. Apostolico, *On context constrained squares and repetitions in a string*, RAIRO Inform. Théor. Appl., 18 (1984), pp. 147–159.

[2] ———, *Optimal parallel detection of squares in strings*, Algorithmica, 8 (1992), pp. 285–319.

[3] A. Apostolico, D. Breslauer, and Z. Galil, *Optimal parallel algorithms for periods, palindromes and squares*, in Proc. 19th International Colloquium on Automata, Languages, and Programming, Lecture Notes in Comput. Sci. 623, Springer-Verlag, Berlin, 1992, pp. 296–307.

[4] A. Apostolico and F. P. Preparata, *Optimal off-line detection of repetitions in a string*, Theoret. Comput. Sci., 22 (1983), pp. 297–315.

[5] P. Beame and J. Håstad, *Optimal bound for decision problems on the CRCW-PRAM*, J. Assoc. Comput. Mach., 36 (1989), pp. 643–670.

[6] D. R. Bean, A. Ehrenfeucht, and G. F. McNulty, *Avoidable patterns in strings of symbols*, Pacific J. Math., 85 (1979), pp. 261–294.

[7] J. Berstel, *Sur les mots sans carré définis par un morphism*, in Proc. 6th International Colloquium on Automata, Languages, and Programming, Lecture Notes in Comput. Sci. 71, Springer-Verlag, Berlin, 1979, pp. 16–25.

[8] R. P. Brent, *Evaluation of general arithmetic expressions*, J. Assoc. Comput. Mach., 21 (1974), pp. 201–206.

[9] D. Breslauer, *Efficient string algorithmics*, Ph.D. thesis, Department of Computer Science, Columbia University, New York, 1992.

[10] D. Breslauer and Z. Galil, *An optimal $O(\log\log n)$ time parallel string matching algorithm*, SIAM J. Comput., 19 (1990), pp. 1051–1058.

[11] ———, *A lower bound for parallel string matching*, SIAM J. Comput., 21 (1992), pp. 856–862.

[12] ———, *Finding all periods and initial palindromes of a string in parallel*, Algorithmica, 14 (1995), pp. 355–366.

[13] M. Crochemore, *An optimal algorithm for computing the repetitions in a word*, Inform. Process. Lett., 12 (1981), pp. 244–250.

[14] ———, *Sharp characterizations of squarefree morphisms*, Inform. Process. Lett., 18 (1982), pp. 221–226.

[15] ———, *Transducers and repetitions*, Theoret. Comput. Sci., 12 (1986), pp. 63–86.

[16] M. Crochemore and W. Rytter, *Efficient parallel algorithms to test square-freeness and factorize strings*, Inform. Process. Lett., 38 (1991), pp. 57–60.

[17] ———, *Usefulness of the Karp–Miller–Rosenberg algorithm in parallel computations on strings and arrays*, Theoret. Comput. Sci., 88 (1991), pp. 59–82.

[18] ———, *Periodic prefixes in texts*, in Proc. 1991 Sequences Workshop: Sequences II: Methods in Communication, Security and Computer Science, R. Capocelli, A. De Santis, and U. Vaccaro, eds., Springer-Verlag, Berlin, 1993, pp. 153–165.

[19] F. E. Fich, R. L. Ragde, and A. Wigderson, *Relations between concurrent-write models of parallel computation*, SIAM J. Comput., 17 (1988), pp. 606–627.

[20] Z. Galil, *String matching in real time*, J. Assoc. Comput. Mach., 28 (1981), pp. 134–149.

[21] Z. Galil and J. Seiferas, *Time-space-optimal string matching*, J. Comput. System Sci., 26 (1983), pp. 280–294.

[22] M. Harrison, *Introduction to Formal Language Theory*, Addison–Wesley, Reading, MA, 1978.

[23] M. Lothaire, *Combinatorics on Words*, Addison–Wesley, Reading, MA, 1983.

[24] R. C. LYNDON AND M. P. SCHÜTZENBERGER, *The equation $a^m = b^n c^p$ in a free group*, Michigan Math. J., 9 (1962), pp. 289–298.

[25] G. M. MAIN AND R. J. LORENTZ, *An $O(n \log n)$ algorithm for finding all repetitions in a string*, J. Algorithms, 5 (1984), pp. 422–432.

[26] ———, *Linear time recognition of squarefree strings*, in Combinatorial Algorithms on Words, A. Apostolico and Z. Galil, eds., NATO ASI Ser. F 12, Springer-Verlag, Berlin, 1985, pp. 271–278.

[27] M. O. RABIN, *Discovering repetitions in strings*, in Combinatorial Algorithms on Words, A. Apostolico and Z. Galil, eds., NATO ASI Ser. F 12, Springer-Verlag, Berlin, 1985, pp. 271–278.

[28] R. ROSS AND R. WINKLMANN, *Repetitive strings are not context-free*, Technical report CS-81-070, Washington State University, Pullman, WA, 1981.

[29] A. THUE, *Über unendliche zeichenreihen*, Norske Vid. Selsk. Skr. Mat. Nat. Kl. (Cristiania), 1906 (7), pp. 1–22.

[30] ———, *Über die gegenseitige lage gleicher teile gewisser zeichenreihen*, Norske Vid. Selsk. Skr. Mat. Nat. Kl. (Cristiania), 1912 (1), pp. 1–67.

SIAM J. COMPUT.
Vol. 25, No. 6, pp. 1332–1357, December 1996

012

THE WAKEUP PROBLEM*

MICHAEL J. FISCHER[†], SHLOMO MORAN[‡], STEVEN RUDICH[§], AND GADI TAUBENFELD[¶]

Abstract. We study a new problem—the *wakeup problem*—that seems to be fundamental in distributed computing. We present efficient solutions to the problem and show how these solutions can be used to solve the consensus problem, the leader-election problem, and other related problems. The main question we try to answer is "How much memory is needed to solve the wakeup problem?" We assume a model that captures important properties of real systems that have been largely ignored by previous work on cooperative problems.

Key words. fault tolerance, shared memory, concurrency, algorithms

AMS subject classifications. 68M99, 68Q10, 68Q25

1. Introduction.

1.1. The wakeup problem.

The *wakeup problem* is a deceptively simple new problem that seems to be fundamental in distributed computing. The goal is to design a t-resilient protocol for n asynchronous processes in a shared-memory environment such that at least p processes eventually learn that at least τ processes have waked up and begun participating in the protocol. Put another way, the wakeup problem with parameters n, t, τ, and p is to find a protocol such that in any fair run of n processes with at most t failures, at least p processes eventually *know* that at least τ processes have taken at least one step in the past. The only kind of failures we consider are crash failures, in which a process may become faulty at any time during its execution, and when it fails, it simply stops participating in the protocol.

In the wakeup problem, it is known a priori by all processes that at least $n - t$ processes will eventually wake up. The goal is simply to have a point in time at which the fact that at least τ processes have already woken up is *known* to p processes. It is not required that this time be the earliest possible, and faulty processes are included in the counts of processes that have woken up and that know about that fact. Note that in a solution to the wakeup problem, at least $p - t$ correct processes eventually learn that at least $\tau - t$ correct processes are awake and participating in the protocol.

The significance of this problem is twofold. First, it seems generally useful to have a protocol such that after a crash of the network or a malicious attack, the remaining correct processes can figure out if sufficiently many other processes remain active to carry out a given task. Second, a solution to this problem is a useful building block for solving other important problems such as the consensus [Abr88, Fis83, PSL80], leader-election [FL87, Pet82], memory-initialization [Hem89], phase-synchronization [Mis91], and processor-identity [LP90] problems.

* Received by the editors September 7, 1993; accepted for publication (in revised form) March 30, 1995. A preliminary version of this work appeared in *Proc. 22nd Annual Symposium on Theory of Computing*, Association for Computing Machinery, New York, 1990. This research was supported in part by ONR contract N00014-89-J-1980, NSF grant CCR-8405478, the Hebrew Technical Institute scholarship, the Technion V.P.R. Funds–Wellner Research Fund, and the Foundation for Research in Electronics, Computers, and Communications, administrated by the Israel Academy of Sciences and Humanities.

[†] Computer Science Department, Yale University, New Haven, CT 06520.

[‡] Computer Science Department, Technion, Haifa 32000, Israel.

[§] Computer Science Department, Carnegie Mellon University, Pittsburgh, PA 15213.

[¶] AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974 and Open University of Israel, Tel Aviv 61392, Israel.

1.2. A new model. Much work to date on fault-tolerant parallel and distributed systems has been generous in the class of faults considered but rather strict in the requirements on the system itself. Problems are usually studied in an underlying model that is fully synchronous, provides each process with a unique name that is known to all other processes, and is initialized to a known state at time zero. We argue that none of these assumptions is realistic in today's computer networks, and achieving them even within a single parallel computer is becoming increasingly difficult and costly. Large systems do not run off of a single clock and hence are not synchronous. Providing processes with unique id's is costly and difficult and greatly complicates reconfiguring the system. Finally, simultaneously resetting all of the computers and communication channels in a large network to a known initial state is virtually impossible and would rarely be done even if it were possible because of the large destructive effects it would have on ongoing activities.

Our new model of computation makes none of these assumptions. It consists of a fully asynchronous collection of n identical anonymous processes that communicate via a *single* finite-sized shared register which is initially in an arbitrary unknown state. Access to the shared register is via atomic "read–modify–write" instructions which, in a single indivisible step, read the value in the register and then write a new value that can depend on the value just read. (When only atomic read and atomic write instructions are assumed, the wakeup problem cannot be solved even when $t = 0$, $\tau = 2$, and $p = 1$ since no process can ever learn that the others are awake if the processes are scheduled in a round-robin fashion.)

Assuming an arbitrary unknown initial state relates to the notion of self-stabilizing systems defined by Dijkstra [Dij74]. However, Dijkstra considers only nonterminating control problems such as the mutual-exclusion problem, whereas we show how to solve decision problems such as the wakeup, consensus, and leader-election problems, in which a process makes an irrevocable decision after a finite number of steps.

Before proceeding, we should address two possible criticisms of shared-memory models in general and our model in particular. First, most computers implement only reads and writes to memory, so why do we consider atomic read–modify–write instructions? One answer is that large parallel systems access shared memory through a communication network which may well possess independent processing power that enables it to implement more powerful primitives than just simple reads and writes. Indeed, such machines have been seriously proposed [GGK$^+$83, Pea85]. Another answer is that part of our interest is in exploring the boundary between what can and cannot be done, and a proof of impossibility for a machine with read–modify–write access to memory shows a fortiori the corresponding impossibility for the weaker read/write model.

A second possible criticism is that real distributed systems are built around the message-passing paradigm and that shared-memory models are unrealistic for large systems. Again, we have several possible answers. First, the premise may not be correct. Experience is showing that message-passing systems are difficult to program, so increasing attention is being paid to implementing shared-memory models, either in hardware (e.g., the Fluent machine [RBJ88]) or in software (e.g., the Linda system [CG89]). Second, message-passing systems are themselves an abstraction that may not accurately reflect the realities of the underlying hardware. For example, message-passing systems typically assume infinite buffers for incoming messages, yet nothing is infinite in a real system, and indeed overflow of the message buffer is one kind of fault to which real systems are subject. It is difficult to see how to study a kind of fault which is assumed away by the model. Finally, at the lowest level, communication hardware looks very much like shared memory. For example, a wire from one process to another can be thought of as a binary shared register which the first process can write (by injecting a voltage) and the second process can read (by sensing the voltage).

1.3. Space-complexity results. The main question we try to answer is "How many values v for the shared register are necessary and sufficient to solve the wakeup problem?" The answer both gives a measure of the communication-space complexity of the problem and also provides a way of assessing the cost of achieving reliability. We give a brief overview of our results below.

1.3.1. Fault-free solutions. First, we examine what can be done in the absence of faults (i.e., $t = 0$). We present a solution to the wakeup problem in which one process learns that all other processes are awake (i.e., $p = 1$ and $\tau = n$), and it uses a single 4-valued register (i.e., $v = 4$). The protocol for achieving this is quite subtle and surprising. It can also be modified to solve the leader-election problem. Based on this protocol, we construct a fault-free protocol that reaches consensus on one out of k possible values using a 5-valued register. Finally, we show that there is no fault-free solution to the wakeup problem with only two values (i.e., one bit) when $\tau \geq 3$.

1.3.2. Fault-tolerant solutions: Upper bounds. We start by showing that the fault-free solution which uses a single 4-valued register, mentioned in the previous section, can actually tolerate t failures for any $\tau \leq ((2n-2)/(2t+1)+1)/2$. Using many copies of this protocol, we construct a protocol with $v = 8^{t+1}$ that tolerates t faults when $\tau \leq n - t$. Thus, if t is a constant, then a constant-sized shared memory is sufficient, independent of n. However, the constant grows exponentially with t. An easy protocol exists with $v = n$ that works for any t and $\tau \leq n - t$. This means that the above exponential result is only of interest for $t \ll \log n$. Finally, we show that for any $t < n/2$, there is a t-resilient solution to the wakeup problem for any $\tau \leq \lfloor n/2 \rfloor + 1$ using a single $O(t)$-valued register.

1.3.3. Fault-tolerant solutions: A lower bound. We prove that for any protocol P that solves the wakeup problem for parameters n, t, and τ, where $1 < t \leq 2n/3$ and $\tau > \lceil n/3 \rceil$, and for every $0 < \varepsilon \leq 1/2$, the number of shared-memory values used by P is at least $(W+1)^{\alpha}$, where $W = \varepsilon t^2/(2(n-t))$ and $\alpha = 1/(\log_2((n-t)/((1-\varepsilon)t)+2))$. The proof is quite intricate and involves showing that for any protocol with too few memory values, there is a run in which $n - t$ processes wake up and do not fail, yet no process can distinguish that run from another in which fewer than τ wake up; hence no process knows that τ are awake.

When we take t to be a constant fraction of n, we get the following immediate corollary. Let P be a protocol that solves the wakeup problem for parameters n, t, and τ, where $t \geq n/c$ and $\tau > \lceil n/3 \rceil$. Let V be the set of shared-memory values used by P. Let $\gamma = 1/(\log_2(c+1))$ and $\delta > 0$. Then $|V| = \Omega(n^{\gamma-\delta})$. The corollary gives the bound that we obtain in the case where $t = \Omega(n)$. However, when $t = O(n^{\xi})$ for a constant $\xi < 1$ we get that our lower bound $(W+1)^{\alpha} = O(1)$ and hence is not interesting.

1.4. Relation to other problems. We establish connections between the wakeup problem and two fundamental problems in distributed computing: the consensus problem and the leader-election problem. These two problems lie at the core of many problems for fault-tolerant distributed applications [Abr88, AG85, CR79, DDS87, DKR82, DLS88, Fis83, FL87, FLM86, FLP85, HS80, KKM, KMZ84, Pet82, PKR84, PSL80, TKM89a, Tau91].

We show that (1) any protocol that uses v values and solves the wakeup problem for $t < n/2$, $\tau > n/2$, and $p = 1$ can be transformed into t-resilient consensus and leader-election protocols which use $8v$ values and (2) any t-resilient consensus or leader-election protocol that uses v values can be transformed into a t-resilient protocol which uses $4v$ values and solves the wakeup problem for any $\tau \leq \lfloor n/2 \rfloor + 1$ and $p = 1$.

Using the first result above, we can construct efficient solutions to both the consensus and leader-election problems from solutions for the wakeup problem. The second result implies that the lower bound proved for the wakeup problem holds for these other two problems. As a consequence, the consensus and the leader-election problems are space equivalent in our model.

2. Definitions and notations.

2.1. Protocols and knowledge. An n-process *protocol* $P = (C, N, R)$ consists of a nonempty set C of *runs*, an n-tuple $N = (q_1, \ldots, q_n)$ of *process id's* (or *processes* for short), and an n-tuple $R = (R_1, \ldots, R_n)$ of sets of *registers*. Informally, R_i includes all the registers that process q_i can access. We assume throughout this paper that $n \geq 2$.

A *run* is a pair (f, S) where f is a function which assigns initial values to the registers in $R_1 \cup \cdots \cup R_n$ and S is a finite or infinite sequence of events. (When S is finite, we also say that the run is finite.) An *event* $e = (q_i, v, r, v')$ means that process q_i in one atomic step first reads a value v from register r and then writes a value v' into register r. We say that the event e *involves* process q_i and register r and that process q_i performs a *read–modify–write* operation on register r.

The *value* of a register at a finite run is the last value that was written into that register or is its initial value if no process wrote into the register. We use $value(r, \rho)$ to denote the value of r at a finite run ρ.

A register r is said to be *local* if there exists an i such that $r \in R_i$ and for any $j \neq i, r \notin R_j$. A register is *shared* if it is not local. In this paper, we restrict our attention to protocols which have exactly one register which is shared by all the processes (i.e., $|R_1 \cap \cdots \cap R_n| = 1$) and all other registers are local. We assume that all local registers of process q_i $(1 \leq i \leq n)$ have names of the form $r.i$. Furthermore, we assume that for any two processes q_i and q_j, the (local) register $r.i$ exists iff the register $r.j$ exists.

If S' is a prefix of S, then the run (f, S') is *a prefix* of (f, S) and (f, S) is *an extension* of (f, S'). Let $\langle S; S' \rangle$ be the sequence obtained by concatenating the sequences S and S'. For a run $\rho = (f, S)$, let $\langle \rho; S' \rangle$ be an abbreviation for $(f, \langle S; S' \rangle)$. For any sequence S, let S_i be the subsequence of S containing all events in S which involve q_i. Runs (f, S) and (f', S') are *equivalent with respect to* q_i, denoted by $(f, S) \overset{i}{\sim} (f', S')$, iff $S_i = S'_i$. Let *null* denote the empty sequence.

The set of runs of each protocol considered in this paper is assumed to satisfy the following five properties.

- ρ is a run iff every prefix of ρ is a run.
- Let ρ and ρ' be finite runs such that $\rho \overset{i}{\sim} \rho'$ and $value(r, \rho) = value(r, \rho')$. Then $\langle \rho; (q_i, v, r, v') \rangle$ is a run iff $\langle \rho'; (q_i, v, r, v') \rangle$ is a run. That is, if some event can happen at a process q_i at some point in a run, then the same event can happen at any run that is equivalent to that run w.r.t. q_i provided that the register q_i accesses in that event has the same value in both run.
- Let $\langle \rho; (q_i, v, r, v') \rangle$ be a run. Then $v = value(r, \rho)$. That is, it is possible to

read only the last value that is written into a register.

- Let r be the single shared register. For any run ρ, there exists a run $(g, null)$, where $g(r) = value(r, \rho)$. That is, nothing can be assumed about the initial values.
- Let π be a permutation of $\{1, \ldots, n\}$, let S_π be the sequence of events S where for every $1 \leq i \leq n$ every appearance of q_i in S is replaced by $q_{\pi(i)}$, and let f_π be a function where $f_\pi(r) = f(r)$ for the shared register r and $f_\pi(r.\pi(i)) = f(r)$ for any local register $r.i$. Then if (f, S) is a run then (f_π, S_π) is also a run for every permutation π. That is, the processes are anonymous and identically programmed.

Notice that the above properties allow nondeterministic processes. However, for convenience, we will assume that processes are deterministic.

We are now ready to define the notion of knowledge in a shared-memory environment. In the following, we use *predicate* to mean a set of runs.

DEFINITION. *For a process q_i, predicate b, and finite run ρ,* process q_i knows b at ρ *iff for all ρ' such that $\rho \overset{i}{\sim} \rho'$, it is the case that $\rho' \in b$.*

We say that a process p learns *a predicate b in a run ρ if p knows b in ρ but it does not know b in any proper prefix of ρ.*

For simplicity, we assume that a process always takes a step whenever it is scheduled. A process that takes infinitely many steps in a run is said to be *correct* in that run; otherwise, it is *faulty*. We say that an infinite run is *l-fair* iff at least l processes are correct in it.

2.2. Wakeup, consensus, and leader-election protocols. In this subsection, we formally define the notions of t-resilient wakeup, consensus, and leader-election protocols $(0 \leq t \leq n)$. We say that a process q_i is *awake* in a run if the run contains an event that involves q_i. The predicate "*at least τ processes are awake*" is the set of all runs for which there exist τ different processes which are awake in the run. Note that a process that fails after taking a step is nevertheless considered to be awake in the run.

- A *wakeup protocol* with parameters n, t, τ, and p is a protocol for n processes such that for any $(n - t)$-fair run ρ, there exists a finite prefix of ρ in which at least p processes *know* that at least τ processes are awake in ρ.

It is easy to see that a wakeup protocol exists only if $\max(p, \tau) \leq n - t$, and hence, from now on, we assume that this is always the case. We also assume that $\min(p, \tau) \geq 1$.

In the following, whenever we speak about a solution to the wakeup problem without mentioning p, we are assuming that $p = 1$.

- A *t-resilient k-consensus protocol* is a protocol for n processes where each process has a local read-only input register and a local write-once output register. For any $(n - t)$-fair run, there exists a finite prefix in which all the correct processes decide on some value from a set of size k (i.e., each correct process writes a *decision value* into its local output register), the decision values written by all processes are the same, and the decision value is equal to the input value of some process.

In the following, whenever we say "consensus" (without mentioning specific k), we mean "binary consensus," where the possible decision values are 0 and 1.

- Let P be a protocol for n processes, where each process has a local write-once output register, and let ρ be a finite run of P. We say that a process q_i *commits* to a value $v \in \{0, 1\}$ in ρ if q_i either has already written or eventually writes v to its output register in any $(n - t)$-fair extension of ρ in which q_i is correct. A process q_i is *elected* in ρ if q_i *knows* that it is committed to 1 in ρ. P is said to be a *t-resilient leader-election protocol* if in any $(n - t)$-fair run of P, there exists a finite prefix in which exactly one process is elected and all other processes (correct or faulty) commit to the value 0. The elected process is called the *leader.*

Notice that here we need to use the notions "commits" and "elected" rather than "decides" since the elected leader might fail just before it writes to its output register (at which point it knows that it is committed). Also, we observe that because processes are identical and anonymous, there can be an a priori leader (if one process is elected without taking any steps, then all processes do so). Thus a process is elected in a run only if it participates in this run—a fact which is used in the sequel.

3. Fault-free solutions. In this section, we develop the *seesaw protocol*, which solves the fault-free wakeup problem using a single 4-valued shared register. Then we show how the seesaw protocol can be used to solve the k-valued consensus problem. Finally, we show that it is impossible to solve the wakeup problem using only one shared bit.

To understand the seesaw protocol, the reader should imagine a playground with a seesaw in it. The processes will play the protocol on the seesaw, adhering to strict rules. When each process enters the playground (wakes up), it sits on the up side of the seesaw, causing it to swing to the ground. Only a process on the ground (or down side) can get off, and when it does, the seesaw must swing to the opposite orientation. These rules enforce a balance invariant which says that the number of processes on each side of the seesaw differs by at most one (the heavier side always being down).

Each process enters the playground with two tokens. The protocol will force the processes on the bottom of the seesaw to give away tokens to the processes on the top of the seesaw. Thus token flow will change direction depending on the orientation of the seesaw. Tokens can be neither created nor destroyed. The idea of the protocol is to cause tokens to concentrate in the hands of a single process. A process that sees $2k$ tokens knows that at least k processes are awake. Hence if it is guaranteed that eventually some process will see at least 2τ tokens, the protocol is by definition a wakeup protocol with parameter τ, even if the process does not know the value of τ and hence does not know when the goal has been achieved.

Following is the complete description of the seesaw protocol. The 4-valued shared register is easily interpreted as two bits which we call the "token bit" and the "seesaw" bit. The two states of the token bit are called "token present" and "no token present." We think of a public *token slot* which either contains a token or is empty, according to the value of the token bit. The two states of the seesaw bit are called "left side down" and "right side down." The "seesaw" bit describes a virtual seesaw which has a left and a right side. The bit indicates which side is down (implying that the opposite side is up).

Each process remembers in private memory the number of tokens it currently possesses and which of four states it is currently in with respect to the seesaw: "never been on," "on left side," "on right side," and "got off." A process is said to be on the up side of the seesaw if it is currently "on left side" and the seesaw bit is in state "right side down" or if it is currently "on right side" and the seesaw bit is in state "left side down." A process initially possesses two tokens and is in state "never been on."

We define the protocol by a list of rules. When a process is scheduled, it looks at the shared register and at its own internal state and carries out the first applicable rule, if any. If no rule is applicable, it takes a null step which leaves its internal state and the value in the shared register unchanged.

Rule 1 (start of protocol). This rule is applicable if the scheduled process is in state "never been on." The process gets on the up side of the seesaw and then flips the seesaw bit. By "get on," we mean that the process changes its state to "on left side" or "on right side" according to whichever side is up. Since flipping the seesaw bit causes that side to go down, the process ends up on the down side of the seesaw.

Rule 2 (emitter). This rule is applicable if the scheduled process is on the down side of the seesaw and has one or more tokens and the token slot is empty. The process flips the token bit (to indicate that a token is present) and decrements by one the count of tokens it possesses. If its token count thereby becomes zero, the process flips the seesaw bit and gets off the seesaw by setting its state to "got off."

Rule 3 (absorber). This rule is applicable if the scheduled process is on the up side of the seesaw and a token is present in the token slot. The process flips the token bit (to indicate that a token is no longer present) and increments by one the count of tokens it possesses.

Note that if a scheduled process is on the down side and has $2k-1$ tokens and a token is present in the token slot, then, although no rule is applicable, the process nevertheless sees a total of $2k$ tokens and hence knows that k processes have woken up.

The two main ideas behind the protocol can be stated as invariants.

Token invariant. The number of tokens in the system is either $2n$ or $2n+1$ and does not change at any time during the protocol. (The number of tokens in the system is the total number of tokens possessed by all of the processes plus 1 if a token is present in the token bit slot.)

Proof. The number of tokens in the starting configuration is $2n$ with the possible addition of one token present in the token bit slot. The rules that effect tokens are Rules 2 and 3, both of which maintain the token invariant.

Balance invariant. The number of processes on the left and right sides of the seesaw is either perfectly balanced or favors the down side of the seesaw by one process.

Proof. The seesaw starts empty, zero on either side. Rule 1 preserves the invariant because a process gets on the up side and then flips the seesaw. If a process runs out of tokens, it must be on the down side of the seesaw; hence when Rule 2 is applied, the invariant is maintained.

THEOREM 3.1. *Let $t = 0$. The seesaw protocol uses a 4-valued shared register and is a wakeup protocol for n, t, and τ (and $p = 1$), where n and τ are arbitrary and $t = 0$. (Note that the rules for the protocol do not mention n or τ.)*

Proof. By the token invariant, there are no more than $2n+1$ tokens in the system. At most two come from each player; at most one comes from the initialized state of the token bit. Hence if a process sees 2τ tokens, it has to be the case that at least τ processes are awake.

Next, we argue that the protocol comes to a state where everybody has awakened and there is only one process remaining on the seesaw. We know there will be a time when everybody is awake. Furthermore, for any number of processes $m \geq 2$ still active on the seesaw, there will be a future time when there are only $m-1$ processes on the seesaw. By the balance invariant, there are some processes on both sides and hence eventually either Rule 2 or Rule 3 is applicable (i.e., there is no deadlock). Each process has awakened; hence Rule 1 will no longer apply. Applying Rules 2 and 3 will cause tokens to flow from the down side to the up side; eventually, the token count of a down-side process will become zero and the process will get off the seesaw. Hence there will eventually be only one process remaining on the seesaw. This process will see $2n$ tokens and will know that all other processes are awake. □

In applications of wakeup protocols, it is often desirable for the processes to know the value of τ so that a process learning that τ processes are awake can stop participating in the wakeup protocol and take some action based on that knowledge. The seesaw protocol can be easily modified to have this property by adding a termination rule immediately after Rule 1.

Rule 1a (end of protocol). This rule is applicable if the scheduled process is on the seesaw and sees at least 2τ tokens, where the number of tokens that the process sees is the number it possesses plus one if a token is present in the token slot. The process thus knows that τ processes have woken up. It gets off the seesaw (i.e., terminates) by setting its state to "got off."

The seesaw protocol can also be used to solve the leader-election problem by electing the first process that sees $2n$ tokens. By adding a fifth value, everyone can be informed that the leader was elected, and the leader can know that everyone knows. Now the leader can transmit an arbitrary message—for example, a consensus value—to all the other processes without using any more new values through a kind of serial protocol. This leads to our next theorem.

THEOREM 3.2. *In the absence of faults, it is possible to reach consensus on one of k values using a single* 5*-valued shared register.*

Proof. Assume that the processes have been running the seesaw protocol in which each process initially has two tokens. A process becomes leader when it accumulates $2n$ tokens, at which time possibly one more token remains elsewhere in the system.

Let *end* be a fifth value. The leader now puts *end* in the shared register. Any process that sees *end* for the first time replaces it with (*no token present, left side down*). The leader repeats this $n - 1$ times, waiting each time for *end* to be removed from the register, after which time each other process knows of the existence of the leader. When the leader notices the last *end* disappear, it knows that everyone knows.

Note that at the start of these exchanges, all other processes save one are out of tokens and will ignore all messages except *end.* Any nonleader process possessing a single token will either ignore the message (*no token present, left side down*) or will change it to (*token present, right side down*), depending on its type bit, and thereafter ignore all messages except *end.* Thus from the time the leader is elected until the time that everyone knows of its election, the only possible shared register values are *end*, (*no token present, left side down*), and (*token present, right side down*). Hence the remaining two values, (*no token present, right side down*) and (*token present, left side down*), can be reused to initiate sending the message because they will not appear until after the leader knows that everyone knows. Call these values *data*1 and *ack*1, respectively. Call values (*no token present, left side down*) and (*token present, right side down*) *data*2 and *ack*2, respectively.

Let $1 \leq m \leq k$ be the consensus value, which we take to be the leader's initial value. Here is the protocol that the leader now uses to send m to all other processes. The leader executes m data phases. Each process counts the number of data phases executed. At the end of the m phases, the leader terminates the protocol by putting *end* back in the register. Each process terminates when it sees *end*, in which case it also knows the number of phases and hence the consensus value.

The first data phase involves the leader putting *data*1 into memory $n - 1$ times. Each follower process, upon seeing *data*1, replaces it with *ack*1, increments its phase counter, and enters the next phase, where it waits for *data*2 or *end.* The second phase uses *data*2 and *ack*2, and subsequent phases alternate between the two versions of the values, odd-numbered phases using *data*1 and *ack*1 and even-numbered phases using *data*2 and *ack*2. □

Finally, we claim that the seesaw protocol cannot be improved to use only a single binary register. A slightly weaker result than Theorem 3.3 was also proved by Joe Halpern [Hal]. The question of whether three values suffice was settled affirmatively by Valois in [Va95].

THEOREM 3.3. *There does not exist a solution to the wakeup problem which uses only a single binary register when* $\tau \geq 3$.

In order to prove Theorem 3.3, we first prove a simple lemma. We say that a process writes the value a_∞ if the process writes a and at the infinite extension in which this process is the only one that is activated, a appears infinitely many times. We notice that when the memory is bounded, for any run ρ and any process p, if p is run alone from ρ, then p must eventually write a_∞ for some a.

LEMMA 3.1. *In any wakeup protocol where* $\tau \geq 2$, *if the initial value is a and only one process wakes up and it is activated alone forever, then it will never write* a_∞.

Proof. Assume to the contrary that the lemma does not hold. We show that this leads to a contradiction by constructing an n-fair run in which the initial value of that shared register is a, the value a appears infinitely many times, and yet no process knows that any other process is awake. We construct the run ρ by activating the processes in a round-robin fashion infinitely many times, starting with a as the initial value. Each time a process is scheduled, we let it run until it writes a_∞. Each process cannot distinguish ρ from the run constructed similarly in which it is the only process that is activated. Hence no process ever knows that any other process is awake. □

Proof of Theorem 3.3. We first assume that n is even and construct an n-fair run called ρ such that in each prefix of that run, each process only knows that one other process is awake.

Assume that the initial value is $b \in \{0,1\}$, let q be an arbitrary process, and consider the following scenario. First, q runs alone until it writes a_∞ (by Lemma 3.1, $a \neq b$). At that point, we interfere and flip the shared bit so that its value is again b. Afterwards, we let q continue until it writes a_∞ again and then we flip the bit and so on. Let $flip(b)$ be the number of times that q writes a_∞ at such an infinite run. We consider the two possible cases.

The first case is when both $flip(0)$ and $flip(1)$ are infinite. We construct the run ρ by activating the processes in a round-robin fashion infinitely many times, starting with 0 as the initial value. Each time a process is scheduled, if the value of the shared bit is a (b), we let it run until it writes b_∞ (a_∞). Each process cannot distinguish ρ from the run constructed similarly in which only two processes participate. Hence no process ever knows that more than one other process is awake.

The other case is the negation of the previous one. Assume w.l.o.g. that $flip(0) = k$ for some positive number k and that $flip(0) \leq flip(1)$. We construct the run ρ by first activating the processes in a round-robin fashion exactly as in the previous construction but only for k rounds, starting with 0 as the initial value. After k rounds, the value of the shared bit is 0. We extend this run to an n-fair run by continuing activating the processes in a round-robin fashion, letting each process make one or more steps whenever it is scheduled until it writes 0_∞. (Note that this is always possible since $flip(0) \leq flip(1)$.) As in the previous case, no process can distinguish this run from the run constructed similarly in which only two processes participate. Hence no process ever knows that more than one other process is awake. This completes the proof when n is even.

Assume that n is odd. Let $m = n - 1$. Since m is even, we can construct exactly as before an m-fair run called ρ in which no process ever knows that more than one other process is awake. Let q be the remaining process. We now construct ρ' as follows. We start with 1 as the initial value and let q run until it writes 0_∞ (as is assured by Lemma 3.1). Then, alternately, we let the other m processes run as in ρ until 0 appears; then we let q take one or more steps until 0 appears (this is assured since previously q wrote 0_∞) and so on. Clearly, q cannot distinguish ρ' from a run where it is the only process that is activated, and hence it never knows that any of the other processes is awake. The other processes cannot distinguish ρ' from ρ and hence never know (as in ρ) that more than one other process is awake. □

4. Fault-tolerant solutions.

In this section, we explore solutions to the wakeup problem which can tolerate $t > 0$ process failures. The seesaw protocol, presented in the previous section, cannot tolerate even a single crash failure for any $\tau > n/3$. The reason is that the faulty process may fail after accumulating $2n/3$ tokens, trapping two other processes on one side of the seesaw, each with $2n/3$ tokens. When $\tau \leq n/3$, the seesaw protocol can tolerate at least one failure. As the parameter τ decreases, the number of failures that the protocol can tolerate increases. This fact is captured by the following theorem.

THEOREM 4.1. *The seesaw protocol is a wakeup protocol for n, t, and τ, where*

$$\tau \leq \frac{(2n-1)/(2t+1)+1}{2}.$$

Proof. Failures affect the protocol in two ways. First, tokens possessed by a failed process are lost to the system. Second, failures can disrupt the balance condition on the number of active processes of each type. Thus, after t failures, up to $t(2\tau - 1)$ tokens can be lost, and the number of active processes of each type can differ by up to $t+1$. (If a faulty process accumulates 2τ tokens, it knows that at least τ processes are awake, and the goal of the protocol is achieved.) This implies that when one reaches a stage in which there are either no emitters or no absorbers, there can remain as many as $t+1$ active processes. In order to guarantee termination, we must be assured that at least one of these remaining processes holds at least 2τ tokens. Since the other t active processes can each hold $2\tau - 1$ tokens, the total number of tokens remaining after t failures must be at least $2\tau + t(2\tau - 1)$. (Notice that at the point when one process accumulates 2τ tokens, there is no token in the shared register.) Hence we must have $2n - t(2\tau - 1) \geq (t+1)(2\tau - 1) + 1$. Solving, we get $2n \geq (2t+1)(2\tau-1)+1$, so $\tau \leq ((2n-1)/(2t+1)+1)/2$. □

If one insists that some nonfailing process learns that τ nonfailing processes have woken up, then a process terminates when it collects $2(\tau + t)$ tokens, and each failing process can take at most $2(\tau + t)$ tokens with it (since it stops accumulating tokens when it has that number). Hence we get the inequality $2n - t(2(\tau+t)) \geq (t+1)(2(\tau+t)-1)+1$. Solving, we get $2n \geq 2(2t+1)(\tau+t)-t$, so $\tau \leq (2n+t)/(2(2t+1))-t$. We note that the seesaw protocol can tolerate up to $n/2-1$ *initial failures* [FLP85, TKM89b].

As we can see, the seesaw protocol needs only four values but is very sensitive to failures. Let us define $\psi = \tau/(n-t)$ as the *sensitivity parameter* of a wakeup protocol. Clearly, in the seesaw protocol, when t is a constant fraction of n, the limit of ψ as n goes to infinity is zero. In the rest of this section, we present three t-resilient wakeup protocols. In the first two protocols, $\psi = 1$, but they need n and 8^{t+1} values. In the third protocol, $\psi \geq 1/2$, but it needs only $O(t)$ values.

THEOREM 4.2. *For any $t < n/6$, there is a wakeup protocol which uses a single 8^{t+1}-valued register and works for any $\tau \leq n - t$.*

Proof. The solution is constructed using $t+1$ copies of the seesaw protocol. Before going into details, let us first reexamine the seesaw protocol. Consider the following situation. There are only three processes, and initially there is a token in the shared variable. Let each process make one move. Now there are two emitters and one absorber. If at that point the absorber fails, the other two processes are captured forever in the protocol.

It is not difficult to see that t faulty processes can trap at most $t+1$ other processes in an execution of the seesaw protocol (i.e., if there is a deadlock, then at most $t+1$ correct processes have not yet terminated). The proof of that fact follows from the invariant that the difference between emitters and absorbers is at most one.

Also, we observe that if we have a leader which is guaranteed not to fail, then one bit is sufficient in order for the leader to learn that $n-t$ processes are awake, assuming up to t failures. This goes as follows. When the leader reads 1, it writes 0; otherwise, it waits. When a slave reads 0, it writes 1; otherwise, it waits. Each slave changes the bit exactly two times. When the leader learns that the bit has been changed $2(n-t)-3$ times from 0 to 1, it knows that $n-t$ processes (including itself) are awake. Call this trivial protocol the *leader protocol.*

Using these observations, we are ready to present, for any $t < n/6$, a t-resilient wakeup protocol for $\tau = n-t$, which uses a single 8^{t+1}-valued register. In this protocol, the processes participate in $t+1$ seesaw protocols in a sequential manner. That is, processes get on the ith protocol only after they get off the $(i-1)$st protocol. In addition, all processes participate in $t+1$ leader protocols in parallel. That is, each process participates in one seesaw protocol and in $t+1$ leader protocols at the same time. Each process behaves according to the following rules.

For all $1 \leq i \leq t+1$, we have the following.

- A process that accumulates $n+1$ tokens in the ith seesaw protocol becomes the *leader* of that protocol and takes the role of the leader in the ith leader protocol (and participate as a slave in all other leader protocols).
- At any time, a process that is not a leader at the ith seesaw protocol participates as a slave in the ith leader protocol.
- Once a leader is elected in the ith seesaw protocol, it immediately stops participating in this protocol and participates only in all the $t+1$ leader protocols. (The justification for that is that if the leader never fails, then it will eventually learn that $n-t$ processes are awake. If it does fail, then we can assume w.l.o.g. that it always fails immediately after it is elected.)

This completes the description of the protocol.

The correctness proof is as follows. Since once a process accumulates $n+1$ tokens in the ith seesaw protocol, it stops participating in it, no other process will ever accumulate $n+1$ tokens in this seesaw protocol. Hence at each seesaw protocol, at most one leader is elected, and at each leader protocol, at most one process participates as a leader. The next observation is that if no reliable leader is elected in one of the first t seesaw protocols, then eventually a reliable leader is elected at the $(t+1)$st seesaw protocol. The reason for this is as follows. Assuming that no reliable leader is elected in the first t seesaw protocols implies that t processes already fail, and hence any process that participates in the $(t+1)$st protocol has to be reliable. The total number of processes that can either fail or be trapped in the first t seesaw protocols is at most $3t$. Hence, since $t < n/6$, it follows that more than $n/2$ processes will eventually participate in the $(t+1)$st seesaw protocol and one of them will eventually be elected. Thus eventually a reliable leader is elected and it will learn that $n-t$ processes are awake by participating as a leader in one of the leader protocols. □

We notice that instead of using a single 8^{t+1}-valued shared register, it is possible to use $t+1$ 4-valued registers and $t+1$ binary registers where a process can read–modify–write only one such register at a time. Although the protocol-sensitivity parameter is optimal, its space complexity grows exponentially with t. Notice that when the number of failures t is a constant, one process can learn that $n-t$ processes are awake with a constant number of values.

THEOREM 4.3. *For any $t < n$, there is a wakeup protocol which uses a single n-valued register and works for any $\tau \leq n-t$.*

Proof. The solution uses a single register called the *counter*, whose values are $\{0, \ldots, n-1\}$. Each process initially records the value of the counter and increments it by 1 (mod n). Thereafter, it reads the value of the counter until it finds out that the counter has advanced by at least τ, which implies that at least τ processes are awake. Clearly, at least one reliable process must see this. □

For later reference, we call the above protocol *the counter protocol.* Although the counter-protocol-sensitivity parameter is optimal (i.e., $\psi = 1$), its space requirement seem to be to big when t is small. Hence the protocol of Theorem 4.2 is better than the counter protocol for $t \ll \log(n)$. In our next solution, ψ is not optimal and ranges from 1/2 to 1 depending on the value of t; however, its space complexity is linear in t.

THEOREM 4.4. *For any $t < n/2$, there is a wakeup protocol which uses a single $O(t)$-valued register and works for any $\tau \leq \lfloor n/2 \rfloor + 1$.*

Proof. Let $t < n/10$. In what follows, we describe a wakeup protocol for $\tau = \lfloor n/2 \rfloor + 1$ which uses a single $O(t)$-valued register. The protocol is presented together with its correctness proof. When $t \geq n/10$, we may use the counter protocol mentioned above. The protocol is obtain by using a counter and the seesaw protocol. The size of the counter is n/k, where the value of k is defined later. Each process executes the seesaw protocol and accesses the counter as follows.

- Each process starts with only one token.
- A process increments the counter iff the following hold:
  it is an absorber (in the seesaw protocol),
  it accumulates k tokens, and
  if it is the second time the process tries to increment the counter, then it must be the case that the counter has been incremented at least t times from the first time the process has incremented the counter.
- When a process increments the counter, it erases all the k tokens it holds and continues to participate as an absorber in the seesaw protocol.
- An absorber in the seesaw protocol never collects more than k tokens. (If it has k tokens and cannot increment the counter, then it does nothing until it either becomes an emitter or can increment the counter.)

Once a process learns that the counter has been incremented by more than $n/(2k)$ times, it knows that more than half of the processes are awake.

It remains to decide what the value of k is as a function of n and t. We consider the following observations.

1. If there is a deadlock, then at most $4t+1$ processes are trapped in it ($2t$ by being faulty or absorbers with k tokens that do not fulfill the conditions to increment the counter, and $2t+1$ emitters). Since each process may hold at most k tokens, $(4t+1)k+1$ tokens may be lost. (The 1 is for the token in the shared register.)

2. At the time the first nonfaulty process reads (and remembers) the counter value, the counter has been incremented at most t times from the startup time of the protocol. Hence the kt tokens that are used for these t increments may be lost.

3. From 1 and 2, at most $(5t+1)k+1$ tokens may be lost.

4. To ensure that eventually one correct process will read the counter and will learn that more than $n/2$ processes are awake, it is enough to require that $(5t+1)k+1 < n/2$.

5. In order that the number of values of the counter will be $O(t)$, we should choose k such that $n/k = O(t)$.

Hence, we end with three requirements for k: (1) k is a positive integer, (2) $k < (n-2)/(10t+2)$, and (3) $n/k = O(t)$. From (1), (2), and the fact that t is an integer, it follows that $t < n/10$. Taking $k = \lfloor (n-3)/(10t+2) \rfloor$ is the best choice for k. For the seesaw protocol, we need four values, and for the counter protocol, we need n/k values, which (for large n) is less then $12t$. Hence a single $48t$-valued register suffices for the protocol that we just described. □

In this last protocol, one process learns that $n/2$ other processes are awake. In order for one process to learn that ln processes are awake, we should replace "$(5t+1)k+1 < n/2$" in observation 4 with "$(5t+1)k+1 < (1-l)n$" and get that a single $O(t/(1-l))$-valued register suffices for solving the wakeup problem for $\tau = ln$.

5. A lower bound.

In this section, we establish a lower bound on the number of shared-memory values needed to solve the wakeup problem, where *only one* process is required to learn that τ processes are awake, assuming that t processes may crash fail (i.e., $p = 1$). To simplify exposition, we assume that $1 < t \leq 2n/3$ and $\tau > \lceil n/3 \rceil$. Also, recall that we already assumed that $\tau \leq n - t$. For the rest of this section, let $0 < \varepsilon \leq 0.5$ be fixed (but arbitrary) and let

$$W = \frac{\varepsilon t^2}{2(n-t)}; \qquad \alpha = \frac{1}{\log_2(\frac{n-t}{(1-\varepsilon)t} + 2)}. \tag{1}$$

THEOREM 5.1. *Let P be a protocol that solves the wakeup problem for parameters n, t, and τ. Let V be the set of shared-memory values used by P. Then $|V| > (W+1)^\alpha$.*

When we take t to be a constant fraction of n, we get the following immediate corollary.

COROLLARY 5.1. *Let P be a protocol that solves the wakeup problem for parameters n, t, and τ, where $t \geq n/c$. Let V be the set of shared-memory values used by P. Let $\gamma = 1/(\log_2(c+1))$ and $\delta > 0$. Then $|V| = \Omega(n^{\gamma-\delta})$.*

Theorem 5.1 is immediate if V is infinite, so we assume throughout this section that V is finite. The proof consists of several parts. First, we define a sequence of directed graphs whose nodes are shared-memory values in V. Each component C of each graph in the sequence has a cardinality k_C and a weight w_C. We establish by induction that $w_C < k_C^{1/\alpha} - 1$. Finally, we argue that in the last graph in the sequence, every component C has weight $w_C \geq W$. Hence $|V| \geq k_C > (W+1)^\alpha$.

5.1. Reachability graphs and terminal graphs.

Let V be the alphabet of the shared register. We say that a value $a \in V$ appears m times in a given run if there are (at least) m different prefixes of that run where the value of the shared register is a.

$a \xrightarrow{r} b$ denotes that there exists a run in which *at most* r processes participate, the initial value of the shared register is a, and the value b appears at least once.

$a \overset{r}{\Longrightarrow} b$ denotes that there exists a run in which *exactly* r processes participate, each process that participates takes infinitely many steps, the initial value of the shared register is a, and the value b appears infinitely many times.

Clearly, $a \overset{r}{\Longrightarrow} b$ implies $a \xrightarrow{r} b$ but not vice versa. Also, for every a and for every $r \leq n$, there exists b such that $a \overset{r}{\Longrightarrow} b$.

We use the following graph-theoretic notions. A directed multigraph[1] G is *weakly connected* if the underlying undirected multigraph of G is connected. A multigraph $G'(V', E')$ is a *subgraph* of $G(V, E)$ if $E' \subseteq E$ and $V' \subseteq V$. A multigraph G' is a *component* of a multigraph G if it is a weakly connected subgraph of G and for any edge (a, b) in G, either both a and b are nodes of G' or both a and b are not in G'. A node is a *root* of a multigraph if there is a directed path from every other node in the multigraph to that node. A *rooted graph* (rooted component) is a graph (component) with at least one root.

A *labeled* multigraph is a multigraph together with a label function that assigns a *weight* in $\mathbf{N}$ to each edge of G. The *weight* of a labeled multigraph is the sum of the weights of its edges. We now define the notion of a reachability graph of a given protocol P.

DEFINITION. *Let V be the set of shared-memory values of protocol P. The* reachability graph G of protocol P *is the labeled directed multigraph with node set V and which has an edge from node a to node b labeled with r iff $a \overset{r}{\Longrightarrow} b$ holds.* (*Note that there may be several edges with different labels between the same two nodes. Note also that G is finite since $a \overset{r}{\Longrightarrow} b$ implies that $r \leq n$.*)

DEFINITION. *A graph C is* closed at node a w.r.t. G *if a is in C and for every node b in G, if (a, b) is an edge of G, then b is in C.*

DEFINITION. *A multigraph T is* terminal w.r.t. G *if T is a subgraph of G, all of T's components are rooted, and T has a component C with root a among its minimal-weight components that is closed at node a w.r.t. G.*

In the rest of the section we prove Theorem 5.1 by constructing a multigraph T which is terminal w.r.t. G in which every component with weight w and size k satisfies $k^{1/\alpha} - 1 > w \geq W$.

5.2. Reachability graphs. The reachability graphs are defined for all protocols. We now concentrate on such graphs constructed from wakeup protocols only. We show that when the weight of a rooted subgraph—say C—is sufficiently small, an edge exists with a label q from a root of C to a node not in C and we can bound the size of q.

For later reference, we call the set of the following three inequalities,

(i) $zq + (z-1)w \leq n$,

(ii) $zq \geq n - t$,

(iii) $\max(q, w) < \tau$,

the *zigzag inequalities.* These inequalities play an important role in our exposition.

LEMMA 5.1. *Let G be a reachability graph of a wakeup protocol with parameters n, t, and τ and let C be a rooted subgraph of G with root a and weight w. If there exist positive integers z and q that satisfy the zigzag inequalities, then there exists a node b of G such that $a \overset{q}{\Longrightarrow} b$, and every such node b is not in C.*

Proof. Let z and q be positive integers that satisfy the zigzag inequalities. By (i), $q \leq n$, so there exists b such that $a \overset{q}{\Longrightarrow} b$. We show that $b \notin C$.

Assume to the contrary that $b \in C$. Let ρ be a q-fair run starting from a in which b is written infinitely often. Since b is in C, there is a path from b to a such that the sum of all the labels of edges in that path is at most w and hence $b \xrightarrow{w} a$. This allows us to construct a run with zq nonfaulty processes starting with a as follows:

[1] A multigraph can have several edges from a to b.

Run q processes according to ρ until b is written. Run w processes until a is written. (This must be possible since $b \xrightarrow{w} a$.) Let these w processes fail. Run a second group of q processes according to ρ until b is written. Run a second group of w processes until a is written, and let them fail. Repeat the above until the zth group of q processes have just been run and b has again been written. At this point, zq processes belong to still-active groups and $(z-1)w$ processes have died. If any processes remain, let them now die without taking any steps. Now an infinite run ρ' on the active processes can be constructed by continuing to run the first group according to ρ until b is written again, then doing the same for the second through zth groups, and repeating this cycle forever.

The result is a zq-fair run. Moreover, no reliable process can distinguish this run from ρ, and hence no reliable process ever knows (in ρ') that more than q processes are awake. Also, obviously, no faulty process knows that more than w processes are awake. Since $\max(q, w) < \tau$ and $zq \geq n - t \geq \tau$, this leads to a contradiction to the assumption that the protocol solves the wakeup problem. □

LEMMA 5.2. *Assume that $1 \leq w \leq W$, where w is an integer. Let*

$$q = \left\lceil \frac{w(n-t)}{(1-\varepsilon)t} \right\rceil, \qquad z = \left\lfloor \frac{n-t}{q} \right\rfloor + 1. \tag{2}$$

Then z and q are positive integers that satisfy the zigzag inequalities.

Proof. From the assumptions that $1 \leq w$, $0 < \varepsilon \leq 0.5$, and $t \leq 2n/3$, it follows that $q \geq \lceil 1/(2(1-\varepsilon)) \rceil = 1$, and hence both z and q are positive integers.

To prove inequality (i), observe that from (2), it follows that $z - 1 \leq (n-t)/q$. Thus

$$zq = (z-1)q + q \leq n - t + q. \tag{3}$$

Also, since $1 \leq w \leq W$ and $\varepsilon \leq 0.5$, it follows from (2) that

$$q \leq \left\lceil \frac{\varepsilon t}{2(1-\varepsilon)} \right\rceil \leq \lceil \varepsilon t \rceil. \tag{4}$$

Hence, from (3) and (4), it follows that

$$zq \leq n - t + \lceil \varepsilon t \rceil = n - \lfloor t - \varepsilon t \rfloor. \tag{5}$$

Next, we show that

$$(z-1)w \leq \lfloor t - \varepsilon t \rfloor. \tag{6}$$

Notice that

$$q = \left\lceil \frac{w(n-t)}{(1-\varepsilon)t} \right\rceil \geq \frac{w(n-t)}{(1-\varepsilon)t}. \tag{7}$$

As already mentioned, it follows from (2) that $z - 1 \leq (n-t)/q$; hence by using (7), we get that $z - 1 \leq (1-\varepsilon)t/w$. Since both z and w are integers, $(z-1)w \leq \lfloor t - \varepsilon t \rfloor$. Thus using (5) and (6), we get that inequality (i) is satisfied.

Inequality (ii) is satisfied immediately since by (2), $z > (n-t)/q$.

Finally, we show that inequality (iii) is satisfied. Recall that we assume that $t \leq 2n/3$ and $\tau > \lceil n/3 \rceil$. It follows from these assumptions that $\tau > \lceil t/2 \rceil$. Since $q \leq \lceil \varepsilon t \rceil \leq \lceil t/2 \rceil$, obviously, $q < \tau$. Also, since $w \leq W$ and $t \leq 2n/3$, substituting in (1) gives $w \leq \varepsilon t \leq n/3$, and hence $w < \tau$. □

5.3. Main construction. In this subsection, we first prove that any rooted component of any terminal graph w.r.t. G has *weight* $> W$. Then we use this to construct a subgraph T of G, all of whose rooted components have size $> (W+1)^{\alpha}$.

LEMMA 5.3. *Let G be the reachability graph of a wakeup protocol with parameters n, t, and τ and let T be terminal w.r.t. G. Any rooted component of T has weight $> W$.*

Proof. Assume that the lemma is false. Then T has a minimal-weight component C with root a and weight $0 \leq w \leq W$ such that C is closed at a. If $w = 0$, then $q = 1$ and $z = n - t$ satisfy the zigzag inequalities; otherwise, by Lemma 5.2, there exist positive integers q and z that satisfy the zigzag inequalities. From Lemma 5.1, there is a node b not in C and an edge $a \overset{q}{\Longrightarrow} b$ in G, contradicting the assumption that C is closed at a. □

LEMMA 5.4. *Let G be the reachability graph of a wakeup protocol with parameters n, t, and τ. There exists a subgraph T of G, all of whose rooted components have size $> (W+1)^{\alpha}$.*

Proof. The following procedure constructs T by adding edges one at a time to an *initial* subgraph T_0 of G until Step 2 fails. The *initial* subgraph T_0 consists of all the nodes of G. First, for each node a, we place exactly one outgoing edge $a \overset{1}{\Longrightarrow} b$ in T_0, and then we delete one arbitrary edge from each cycle. We note two facts about T_0: 1. for every edge $a \overset{1}{\Longrightarrow} b$, $a \neq b$; and 2. every component of T_0 is a directed rooted tree. Fact 1 is a simple variant of Lemma 5.1, while fact 2 follows from the observation that every component of T_0 has k vertices and $k-1$ edges for some $k \geq 2$, the out-degree of every vertex is at most one, and it contains no cycles.

At any stage of the construction, every component of the graph built so far will be a directed rooted tree. Added edges always start at a root and end at a node of a different component. After adding an edge (a, b), the components of a and b are joined together into a single component whose root is the root of b's component, and the weight of the new component is the sum of the weights of the two original components plus the label of the edge from a to b.

PROCEDURE FOR ADDING A NEW EDGE TO T.

Step 1. Select an arbitrary component C of minimal weight w with root a.

Step 2. If $w \leq W$, then find the smallest integer q for which there is an edge $a \overset{q}{\Longrightarrow} b$ in G such that b is not in C. This step fails if $w > W$.

Step 3. Place the edge $a \overset{q}{\Longrightarrow} b$ into T.

First note that by Lemma 5.3, if $w \leq W$, then T is not terminal, and hence Step 2 cannot fail. Let T_i be a graph that is constructed after i applications of the above procedure, where T_0 is an initial graph as defined above. Clearly, any such sequence $\{T_0, T_1, \ldots\}$ is finite. Let T_{last} be the last element in this sequence. Then the weight of any component in T_{last} is greater than W.

Let $\beta = 1/\alpha$. We prove by induction on i, the number of applications of the procedure, that for any graph T_i, all of the components of T_i are rooted, and for any rooted component C, it is the case that $w < k^{\beta} - 1$, $k \geq 2$, and $w \geq 1$, where k is the size of C and w is its weight. This together with the fact that eventually every component C has weight greater than W completes the proof.

As discussed before, each component C of T_0 has a root and has size k at least 2. The component C consists of at most $k-1$ edges with label 1, so its weight is at most $k-1$. Hence the base case holds since $\beta > 1$.

Since T_0 is a subgraph of T_i which also includes all nodes of T_i, it follows that the size and weight of any rooted component of T_i are at least 2 and 1, respectively. Now suppose the procedure adds an edge of label q from component C_1 of size k_1 and weight w_1 to component C_2 of size k_2 and weight w_2. By Step 1, the new edge emanates from a minimal-weight component, so $w_1 \leq w_2$. The weight w of the newly formed component is $w_1 + w_2 + q$, and the number of nodes k is $k_1 + k_2$. We now show that $w < k^\beta - 1$.

By Step 2 of the procedure, $w_1 \leq W$. Hence it follows from Lemma 5.2 that there exist positive integers z' and q' that satisfy the zigzag inequalities and $q' = \lceil w_1(n-t)/((1-\varepsilon)t) \rceil$. Hence by Lemma 5.1, there is an edge of label q' from a root of C_1 to C_2. Thus by the minimality of q (the weight of the edge in Step 2), it follows that $q \leq q'$, which implies that $q \leq w_1(n-t)/((1-\varepsilon)t) + 1$. Let $\mathbf{r} = (n-t)/((1-\varepsilon)t) + 1$. Then

$$w = w_1 + w_2 + q \leq \mathbf{r}\, w_1 + w_2 + 1. \tag{8}$$

Let k_1' and k_2' be defined by the equalities $w_1 = k_1'^\beta - 1$ and $w_2 = k_2'^\beta - 1$. Then $k_1' \leq k_2'$. We claim that

$$\begin{aligned} \mathbf{r}\, w_1 + w_2 + 1 &= \mathbf{r}\, (k_1'^\beta - 1) + (k_2'^\beta - 1) + 1 = \mathbf{r}\, k_1'^\beta + k_2'^\beta - \mathbf{r} \\ &< \mathbf{r}\, k_1'^\beta + k_2'^\beta - 1 \leq (k_1' + k_2')^\beta - 1. \end{aligned} \tag{9}$$

Using the fact that $\mathbf{r} > 1$, everything except the rightmost inequality in Equation (9) is immediate. To prove this last inequality, observe that $2^\beta = \mathbf{r} + 1$, hence equality holds for $k_1' = k_2'$. As k_2' is increased to be larger than k_1', the right side increases more rapidly than the left side since $\beta > 1$; hence, the inequality holds. Finally, by the induction hypothesis, $k_1' < k_1$ and $k_2' < k_2$. Hence

$$(k_1' + k_2')^\beta - 1 < (k_1 + k_2)^\beta - 1 = k^\beta - 1. \tag{10}$$

Putting equations (8)–(10) together gives $w < k^\beta - 1$ or, equivalently, $k > (w+1)^\alpha$. This completes the proof of the lemma. □

Theorem 5.1 follows immediately from Lemma 5.4.

5.4. Remarks. Corollary 5.1 gives the bound that we obtain in the case that $t = \Omega(n)$. However, when $t = O(n^\xi)$ for a constant $\xi < 1$, we get that our lower bound $(W+1)^\alpha = O(1)$ and hence is not interesting. One might wonder whether this defect results simply from the various approximations we made in proving Theorem 5.1. This seems not to be the case, but it is rather a limitation of our proof technique. When $t = O(n^\xi)$, the length of the sequence of graphs $\{T_0, T_1, \ldots\}$ is bounded by a constant, so the size of the largest component of the last element T is also a constant. This remains true even if one uses the least q that satisfies the zigzag inequalities rather than the q that is guaranteed by Lemma 5.2. Hence to obtain a nontrivial lower bound in these cases, we will require either a better bound on the value q in Step 2 of the procedure than can be obtained from the zigzag inequalities or else a whole new proof technique. This also leaves open the possibility that Theorem 4.4 can be substantially improved. Finally, observe that the only place where we used the assumption $\tau > \lceil n/3 \rceil$ is in the last paragraph of the proof of Lemma 5.2, where it is used to prove that $\tau > \lceil t/2 \rceil$. Thus our results remain correct under the weaker restriction $\tau > \lceil t/2 \rceil$.

6. Relation to other problems. In this section, we show that there are efficient reductions between the wakeup problem for $\tau = \lfloor n/2 \rfloor + 1$ and the consensus and leader-election problems. Hence the wakeup problem can be viewed as a basic problem that captures the inherent difficulty of these two problems.

LEMMA 6.1. *In any t-resilient consensus (leader-election) protocol, a process decides (is elected) only when at least $t+1$ processes are awake.*

Proof. We first prove the lemma for a consensus protocol. Assume to the contrary that in some consensus protocol, there exist a process q and a run—say ρ—in which q decides and yet no more than t processes participate in ρ. Let us assume w.l.o.g. that in ρ process q decides on 0 and that the value of the shared register is a. Let Q be the set of processes that do not participate in ρ. Clearly, $|Q| \geq n - t$.

We can now construct a new run in which all processes in Q are correct, the initial value of the shared register is a, only processes in Q participate in it, and all processes in Q read the input value 1. Since the protocol can tolerate up to t failures, this run has a prefix—say ρ'—in which all processes decide. The processes that participate in ρ' must decide on the value 1 since this prefix can be extended to a run where all the n processes read the value 1 (and hence according to the definition of the consensus problem must decide on 1). Since the sets of processes which participate in ρ and ρ' are disjoint and the value of the shared register at the end of ρ is the same as its value at the beginning of ρ', the composition $\langle \rho; \rho' \rangle$ is a run. However, this leads to a contradiction since processes decide on both zero and one at the same run.

The proof for a leader-election protocol is similar. Assume to the contrary that in some leader-election protocol, there exist a process q and a run—say ρ—in which q is elected and yet no more than t processes participate in ρ. Let us assume w.l.o.g. that the value of the shared register in ρ is a. Let Q be the set of processes that do not participate in ρ. Clearly, $|Q| \geq n - t$.

We can now construct a run in which all processes in Q are correct, in which the initial value of the shared register is a, and in which only processes in Q participate. Since the protocol can tolerate up to t failures, this run has a prefix—say ρ'—in which some process $q' \neq q$ is elected and writes 1 in its output register. (Here we used the fact that a process can be elected in a run only if it participates in this run.) Clearly, the composition $\langle \rho; \rho' \rangle$ is a run. However, this leads to a contradiction since two processes are elected at the same run. □

The following theorem shows that in order to decide on some value (to be elected) in a t-resilient consensus (leader-election) protocol, it is necessary to first learn that at least $t+1$ processes are awake. It also shows that in certain cases, learning that $t+1$ processes are awake is sufficient for making a decision. The assumption that the processes are symmetric is not used in the proof of that theorem.

THEOREM 6.1. (1) *Any t-resilient consensus (leader-election) protocol is a t-resilient wakeup protocol for any $\tau \leq t+1$ and $p = n - t$ ($p = 1$).* (2) *For any $t < n/2$, there exist t-resilient consensus and leader-election protocols that are not t-resilient wakeup protocols for any $\tau \geq t+2$.*

Proof. We prove the first part of the theorem. Assume to the contrary that for some consensus (leader-election) protocol, there exist a process q and a run in which q decides (q is elected), and q does not know that at least $t+1$ processes are awake. Hence there exist a process q and a run—say ρ—in which q decides (q is elected) and yet not more than t processes participate in ρ. However, this contradicts Lemma 6.1.

We now prove the second part of the theorem. We first prove the second part for a consensus protocol and then explain why it also holds for a leader-election protocol. We show a consensus protocol in which, in certain n-fair runs, all the processes decide on some value and yet no process ever knows (at this run) that more than $t+1$ processors are awake. The protocol uses values (r, x, y, z), where $r \in \{0, \ldots, m-1\}$, $m = n^2 - 2nt$, and x, y, and z each belongs to $\{0, 1\}$. (Note that since $t < n/2$, we have that $m \geq n$.) For integers $0 \leq i, j \leq m-1$, let $[i, j]$ denote the cyclic interval $[i, i \oplus 1, \ldots, j]$, where $\oplus$ denotes addition modulo m. For $k = 0, \ldots, n-1$, the cyclic intervals $[k(n-2t), k(n-2t) \oplus (t+1)]$ are called *critical intervals.* (Note that critical intervals may overlap.)

The three bits are used as follows: x is flipped exactly once when the decision is made, y is used to hold the decision value, z is used to signal processes that wake up after the decision is made that a decision has already been reached, and hence they should decide on the value in y.

Each process initially records the value of r in a local variable *init* and then increments r by 1. Also, it records the value of x and sets $z = 0$. In alternate steps, it polls r, x, and z to see if they have changed. If either x or z has changed, then a decision has already been reached, and the decision value is in y, in which case the process decides on y and thereafter sets $z = 1$ on each step. Otherwise, if it realizes that the cyclic interval $[init, r]$ includes a critical interval, then it becomes the "decider" process. It then stops running the wakeup protocol, chooses its input as the consensus value, and simultaneously flips x, writes the consensus value to y, and sets $r = 1$. All of this is done with a single read–modify–write. Thereafter, it sets $z = 1$ on each step.

The proof that this works is as follows. Until some process becomes the "decider," every process runs the protocol and no process changes x or writes 1 to z. Since the longest cyclic intervals which do not contain a critical interval are of length $n-t-1$, we have that every cyclic interval $[i, i \oplus j]$, where $n-t \leq j \leq n$, contains a critical interval. Therefore, some process eventually polls r such that the cyclic interval $[init, r]$ of this process includes a critical interval. This process becomes a decider. Every other process that woke up before the decision was made will see on its next step that x has changed and hence no such process will also become a decider. Since fewer than $n-t$ processes wake up after the decision has been made and since they are the only ones now affecting r, r is incremented by less than $n-t$ after a decision is made. Since r is set to 1 by the decider, it must be incremented by at least $n-t$ in order for any of these late processes to become a decider. Thus none of these processes becomes a decider, and hence there is a unique decider. It follows that x and y are written to exactly once as desired, so every process that decides on something chooses the same value.

It remains to show that every process decides. Each process that wakes up before the consensus value has been chosen either is the decider or learns the consensus value on its next step thereafter because it will see that x has changed. Since there are more than t such processes, at least one nonfailing process learns the consensus value, and that process writes 1 to z infinitely many times. Since 0 is written to z at most n times, z eventually stabilizes to 1. Thereafter, every process that has not already decided sees $z = 1$ and decides on its next step.

Clearly, the above protocol also solves the leader-election problem since the process called the "decider" is the elected leader and every other process, when it learns that a decision has been made, knows that it cannot be elected. Finally, note that when the initial value of r is $k(n-2t)$ for some k, it is possible to reach a decision when only $t+1$ processes are awake. □

COROLLARY 6.1. *There is no consensus or leader-election protocol that can tolerate $\lceil n/2 \rceil$ failures.*

Proof. Consider a $(n-t)$-fair run in which only $n-t$ processes are awake. By the first part of Theorem 6.1, when a decision is made (a leader is elected) in this run, at least $t+1$ processes are awake. Hence $n-t \geq t+1$, which implies the corollary. □

THEOREM 6.2. *Any protocol that solves the wakeup problem for any $t < n/2$, $\tau > n/2$, and $p = 1$, using a single v-valued shared register, can be transformed into a t-resilient consensus (leader-election) protocol which uses a single $8v$-valued ($4v$-valued) shared register.*

Proof. First, we show a reduction from the consensus problem to the wakeup problem. Suppose the wakeup solution uses values $1, \ldots, v$. The consensus protocol uses values (r, x, y, z), where $r \in \{1, \ldots, v\}$ and x, y, and z each belong to $\{0, 1\}$. The three bits are used as follows: x is flipped exactly once when the decision is made, y is used to hold the decision value, z is used to to signal processes that wake up after the decision is made that a decision has already been reached, and hence they should decide on the value in y.

Each process initially stores the value of x and sets $z = 0$. It then begins running the wakeup protocol. On alternate steps, it polls x and z to see if they have changed. If either has changed, then a decision has already been reached and the decision value is in y, in which case the process abandons whatever else it was doing, decides on y, and thereafter sets $z = 1$ on each step. Otherwise, it continues running the wakeup protocol. If it learns that more than $n/2$ processes have woken up, if x and z still have not changed, then it becomes the "decider" process. It then stops running the wakeup protocol, chooses its input as the consensus value, and simultaneously flips x and writes the consensus value to y. All of this is done with a single read–modify–write. Thereafter, it sets $z = 1$ on each step.

The proof that this works is fairly straightforward and is similar to the proof of the previous theorem. Until some process decides, every process runs the wakeup protocol and no process changes x nor writes 1 to z. Hence some process will eventually learn that more than $n/2$ processes have woken up, and that process will become a decider. Every other process that woke up before the decision was made will see on its next step that x has changed and will abandon the wakeup protocol; hence no such process will become a decider. Since fewer than $n/2$ processes wake up after the decision has been made and since they are the only ones now affecting r, none of them will learn that more than $n/2$ processes have woken up until they see $z = 1$. Hence none of them will become a decider, so there is a unique decider. It follows that x and y are written to exactly once as desired, so every process that decides on something chooses the same value.

It remains to show that every process decides. Each process that wakes up before the consensus value has been chosen either is the decider or learns the consensus value upon seeing that x has changed. Since there are more than $n/2$ of such processes, at least one nonfailing process learns the consensus value, and that process writes 1 to z infinitely many times. Since 0 is written to z at most n times, z eventually stabilizes to 1. Thereafter, every process that has not already decided sees that $z = 1$ and decides on its next step.

Clearly, the above reduction can be used, with minor modifications, as a reduction from the leader-election problem to the wakeup problem. That is so since the process that becomes the "decider" is the elected leader and every other process, when it learns that a decision has been made, knows that it cannot be elected. Finally, the bit y, which is used to hold the decision value, is not needed in this reduction, and hence it is sufficient to have a single $4v$-valued register. □

COROLLARY 6.2. (1) *There is a* $(\lceil n/2\rceil - 1)$*-resilient consensus (leader-election) protocol that uses a single* $8n$*-valued* $(4n$*-valued) shared register.* (2) *For any* $t < n/2$*, there is a* t*-resilient consensus (leader-election) protocol that uses an* $O(t)$*-valued shared register.*

Proof. The proof follows from Theorems 6.2, 4.3, and 4.4. □

The constants in Corollary 6.2 can be improved. In fact, we have designed an $(\lceil n/2\rceil - 1)$-resilient consensus (election) protocol that uses a single $3n$-valued ($2n$-valued) shared register. Next, we show that the converse of Theorem 6.2 also holds. That is, the existence of a t-resilient consensus or leader-election protocol which uses a single v-valued shared register implies the existence of a t-resilient wakeup protocol for $\tau = \lfloor n/2\rfloor + 1$ which uses a single $O(v)$-valued shared register. The idea of the proof is based on the following observation.

LEMMA 6.2.

1. *Let* ρ *and* ρ' *be two runs of the same consensus protocol, where at least one process decides both in* ρ *and in* ρ'*; all the processes in* ρ *have the same input value—say* a*—and when the first process decides (in* ρ*), it writes some value—say* c*—to the shared register; all the processes in* ρ' *have the same input value—say* $b \neq a$*—and the run* ρ' *starts such that* c *is the value of the shared register. Let* n_ρ *(resp.* $n_{\rho'}$*) be the numbers of processes that are awake in* ρ *(resp.* ρ'*) when the first process decides. Then* $n_\rho + n_{\rho'} > n$*.*

2. *Let* ρ *and* ρ' *be two runs of the same election protocol, where some process is elected both in* ρ *and in* ρ'*; when a process is elected in* ρ*, the shared register has some value—say* c*; the run* ρ' *starts with* c *as the initial value of the shared register. Let* n_ρ *(resp.* $n_{\rho'}$*) be the number of processes that are awake in* ρ *(resp.* ρ'*) when a process is elected. Then* $n_\rho + n_{\rho'} > n$*.*

Proof. We start by proving the first part. Assume to the contrary that for some ρ and ρ' as above, $n_\rho + n_{\rho'} \leq n$. We can construct an n-fair run in which initially n_ρ processes behave as in ρ until the first of them decides on a. (Note that according to the definition of the consensus problem, it has to decide on a.) At this point, put long delays on these processes and let different $n_{\rho'}$ processes behave as in ρ' until someone decides on b. This leads to a contradiction since processes decide on different values at the same run.

The proof of the second part is similar. Assume to the contrary that for some ρ and ρ' as above, $n_\rho + n_{\rho'} \leq n$. We can construct a n-fair run in which initially n_ρ processes behave as in ρ until a process is elected. At this point, put long delays on these processes and let different $n_{\rho'}$ processes behave as in ρ' until another process is elected. This leads to a contradiction since two processes are elected at the same run. □

THEOREM 6.3. *Any* t*-resilient protocol that solves the consensus or leader-election problem using a single* v*-valued shared register can be transformed into a* t*-resilient protocol that solves the wakeup problem for any* $\tau \leq \lfloor n/2\rfloor + 1$ *which uses a single* $4v$*-valued shared register.*

Proof. We show only the reduction from the wakeup problem to the consensus problem. The reduction from the wakeup problem to the leader-election problem is almost the same, and the correctness proofs of the two reductions differ only by using different parts of Lemma 6.2. Suppose the consensus solution uses values $1, \ldots, v$. The wakeup solution based on it uses values (r, x), where $r \in \{1, \ldots, v\}$ and $x \in \{0, 1, 2, 3\}$. Informally, this protocol works as follows. The processes run the consensus protocol such that each process considers the value of $x \pmod 2$ as its input. The first process to decide (while simulating the consensus protocol) increments x by 1 (mod 4), and each process that notices that x has been incremented by 1 restarts the simulation with the new input (i.e., $x \pmod 2$). A process that notices that x has been incremented twice or more realizes that at least two such simulations have been completed and, by Lemma 6.2, knows that a majority of processes are awake.

The following is a detailed description. Each process initially stores the value of the shared register (i.e., of r and x). It then begins running the consensus protocol using $x \pmod 2$ as its input. In alternate steps, in a single read–modify–write instruction, it polls r and x and behaves as follows.

- If it notices that x has been incremented twice or more since the very beginning of the simulation, then the process abandons whatever else it was doing and terminates. As we prove later, at that point the process knows that a majority of processes are awake and hence fulfills the requirements of the wakeup problem.
- Otherwise, if x has not been changed since its previous step, then the process continues running the consensus protocol (by taking one more step). Otherwise, it restarts the simulation of the consensus protocol taking $x \pmod 2$ as its input value. In case that by simulating the consensus protocol, the process reaches a decision (in the consensus protocol), then it increments x by 1.

We now give a correctness proof of the reduction. The notion *a round of a run* corresponds to a portion of a run between two successive changes of x. The first round is the portion from the beginning of the run until the first time x is incremented. Note that after the last round (if such a round exists), the processes may still continue running forever but, by definition, the value x is never changed thereafter. A process *participates* in a given round if, during this round, it simulates a step of the consensus protocol.

It should be noted that it is not clear that each round in a given run (of the simulation) corresponds to a possible execution of the consensus protocol since some processes may participate in more than one round. In order for a round to correspond to a possible execution of the consensus protocol, it is sufficient to show that each process that participates in a round does not participate in any previous round in which $x \pmod 2$ had the same value. This clearly holds in the first two rounds. The next two claims show that this holds in the first four rounds. All of the following claims refer to some specific (but arbitrary) infinite run of the reduction in which at least $n - t$ processes are awake.

1. *Let S_i be the set of processes participating in round i. Then $S_1 \cap S_3 = \emptyset$ and $S_2 \cap S_4 = \emptyset$.* A process that participates in round i ($i \in \{1, 2\}$) will notice at round $i + 2$ (if it is given a chance to precede) that x has been incremented twice and hence will terminate without participating in round $i + 2$.

2. *Each of the first four rounds corresponds to a prefix of a run of the consensus protocol.* This claim follows from the previous one since no process participates in both rounds 1 and 3 or in both rounds 2 and 4.

3. *Once a process notices that x has been incremented twice or more, it knows that a majority of processes are awake.* By (2), we know that once x has been incremented twice or more, a simulation of at least two prefixes of runs of the consensus protocol, both satisfying the conditions of Lemma 6.2 have occurred, and hence by Lemma 6.2, a majority of processes have woken up.

4. *In every run, there are at least two rounds.* A process terminates the simulation of the consensus protocol only when it learns that x has been incremented twice or more. Hence at least two rounds are guaranteed to be completed.

5. *There are at most three rounds in each run.* Assume that four rounds are completed in some run. Let n_1, n_2, n_3, and n_4 be the number of processes partic-

ipating in the first, second, third, and fourth rounds, respectively. By (2), all four rounds correspond to possible runs of the consensus protocol in which some process decides. By Lemma 6.2, this implies that $n_1 + n_2 > n$ and $n_3 + n_4 > n$. Hence $n_1 + n_2 + n_3 + n_4 > 2n$. On the other hand, by (1), $n_1 + n_3 \leq n$ and $n_2 + n_4 \leq n$. However, this means that $n_1 + n_2 + n_3 + n_4 \leq 2n$—a contradiction.

6. *Eventually, a nonfaulty process learns that a majority of processes have woken up.* By Lemma 6.1 and (2), in each round, at least $t + 1$ processes participate, and, in particular, a nonfaulty process participates in each round (that may be the same process). We know by (4) and (5) that in each run, x is incremented either two or three times. Hence the nonfaulty process that participates in the first round will eventually notice that x has been incremented two or three times, and by (3), it will know that a majority of processes are awake.

This completes the proof of the theorem. □

We notice that with one additional bit, it is possible to inform everybody that a majority of processes are awake. It follows from Theorem 6.3 that the lower bound that we proved in the previous section for the wakeup problem when $\tau \geq \lfloor n/2 \rfloor + 1$ also applies to the consensus problem.

COROLLARY 6.3. *Let P be a t-resilient consensus or leader-election protocol and let V be the set of shared memory values used by P. Let W and α be defined as in* §5. *Then $|V| \geq (W + 1)^{\alpha}/4$.*

Proof. The proof follows immediately from Theorem 5.1 and Theorem 6.3. □

COROLLARY 6.4. *There is a t-resilient consensus protocol that uses $O(v)$-valued shared register iff there is a t-resilient leader election protocol that uses $O(v)$-valued shared register.*

Proof. The proof follows immediately from Theorem 6.2 and Theorem 6.3. □

7. Conclusions.

We study the new wakeup problem in a new model where all processes are programmed alike, there is no global synchronization, and it is not possible to simultaneously reset all parts of the system to a known initial state.

Our results are interesting for several reasons:

- They give a quantitative measure of the cost of fault tolerance in shared-memory parallel machines in terms of communication bandwidth.
- They apply to a model which more accurately reflects reality.
- They relate recent results from three different active research areas in parallel and distributed computing, namely the following:

results in shared-memory systems [Blo87, DGS88, FLBB89, Her91, Hem89, Lam86, LA87, LF83, LP90, Mis91, Tau89a, TM89, VA86];

the theory of knowledge in distributed systems [CM86, DM86, FHV84, FI86, FI87, FZ88, Hal86, Had87, HF89, HM90, HZ87, KT86, Leh84, Maz89, MT88, MT91, PR85, Mic89, Tut90];

self-stabilizing protocols [BGW89, BP89, Dij74, Dij86, DIM90, Gou87, Kru79, KK90, Tau89b].

- They give a new point of view and enable a deeper understanding of some classical problems and results in cooperative computing.
- They are proved using techniques that will likely have applications to other problems in distributed computing.

Acknowledgments. We thank Joe Halpern for helpful discussions and the anonymous referee for very constructive comments.

REFERENCES

[Abr88] K. Abrahamson, *On achieving consensus using shared memory*, in Proc. 7th ACM Symposium on Principles of Distributed Computing, Association for Computing Machinery, New York, 1988, pp. 291–302.

[AG85] Y. Afek and A. Gafni, *Time and message bounds for election in synchronous and asynchronous complete networks*, in Proc. 4th ACM Symposium on Principles of Distributed Computing, Association for Computing Machinery, New York, 1985, pp. 186–195.

[BGW89] G. M. Brown, M. G. Gouda, and C.-L. Wu, *Token systems that self-stabilize*, IEEE Trans. Comput., 38 (1989), pp. 845–852.

[Blo87] B. Bloom, *Constructing two-writer atomic registers*, in Proc. 6th ACM Symposium on Principles of Distributed Computing, Association for Computing Machinery, New York, 1987, pp. 249–259.

[BP89] J. E. Burns and J. Pachl, *Uniform self-stabilizing rings*, ACM Trans. Programming Lang. Systems, 11 (1989), pp. 330–344.

[CG89] N. Carriero and D. Gelernter, *Linda in context*, Comm. Assoc. Comput. Mach., 32 (1989), pp. 444–458.

[CM86] M. Chandy and J. Misra, *How processes learn*, J. Distrib. Comput., 1 (1986), pp. 40–52.

[CR79] E. Chang and R. Roberts, *An improved algorithm for decentralized extrema-finding in circular configuration of processes*, Comm. Assoc. Comput. Mach., 22 (1979), pp. 281–283.

[DDS87] D. Dolev, C. Dwork, and L. Stockmeyer, *On the minimal synchronism needed for distributed consensus*, J. Assoc. Comput. Mach., 34 (1987), pp. 77–97.

[DGS88] D. Dolev, E. Gafni, and N. Shavit, *Toward a non-atomic era: l-exclusion as a test case*, in Proc. 20th ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1988, pp. 78–92.

[Dij74] E. W. Dijkstra, *Self-stabilizing systems in spite of distributed control*, Comm. Assoc. Comput. Mach., 17 (1974), pp. 643–644.

[Dij86] ———, *A belated proof of self-stabilization*, J. Distrib. Comput., 1 (1986), pp. 5–6.

[DIM90] S. Dolev, A. Israeli, and S. Moran, *Self-stabilization of dynamic systems assuming only read write atomicity*, J. Distrib. Comput., 7 (1993), pp. 3–16.

[DKR82] D. Dolev, M. Klawe, and M. Rodeh, *An $O(n \log n)$ unidirectional distributed algorithm for extrema finding in a circle*, J. Algorithms, 3 (1982), pp. 245–260.

[DLS88] C. Dwork, N. Lynch, and L. Stockmeyer, *Consensus in the presence of partial synchrony*, J. Assoc. Comput. Mach., 35 (1988), pp. 288–323.

[DM86] C. Dwork and Y. Moses, *Knowledge and common knowledge in a Byzantine environment* i: *Crash failures*, in Theoretical Aspects of Reasoning about Knowledge: Proc. 1986 Conference, Morgan Kaufmann, San Mateo, CA, 1986, pp. 149–169.

[FHV84] R. Fagin, J. Y. Halpern, and M. Vardi, *A model theoretic analysis of knowledge*, in Proc. 25th IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1984, pp. 268–278.

[FI86] M. J. Fischer and N. Immerman, *Foundations of knowledge for distributed systems*, in Theoretical Aspects of Reasoning about Knowledge: Proc. 1986 Conference, Morgan Kaufmann, San Mateo, CA, 1986, pp. 171–185.

[FI87] ———, *Interpreting logics of knowledge in propositional dynamic logic with converse*, Inform. Process. Lett., 25 (1987), pp. 175–181.

[Fis83] M. J. Fischer, *The consensus problem in unreliable distributed systems* (*a brief survey*), in Foundations of Computation Theory, M. Karpinsky, ed., Lecture Notes in Comput. Sci. 158, Springer-Verlag, Berlin, New York, Heidelberg, 1983, pp. 127–140.

[FL87] G. Fredrickson and N. Lynch, *Electing a leader in a synchronous ring*, J. Assoc. Comput. Mach., 34 (1987), pp. 98–115.

[FLBB89] M. J. Fischer, N. A. Lynch, J. E. Burns, and A. Borodin, *Distributed FIFO allocation of identical resources using small shared space*, ACM Trans. Programming Lang. Systems, 11 (1989), pp. 90–114.

[FLM86] M. J. Fischer, N. A. Lynch, and M. Merritt, *Easy impossibility proofs for distributed consensus problems*, J. Distrib. Comput., 1 (1986), pp. 26–39.

[FLP85] M. J. Fischer, N. A. Lynch, and M. S. Paterson, *Impossibility of distributed consensus with one faulty process*, J. Assoc. Comput. Mach., 32 (1985), pp. 374–382.

[FZ88] M. J. Fischer and L. D. Zuck, *Reasoning about uncertainty in fault-tolerant dis-*

tributed systems, in Formal Techniques in Real-Time and Fault-Tolerant Systems, M. Joseph, ed., Lecture Notes in Comput. Sci. 331, Springer-Verlag, Berlin, New York, Heidelberg, 1988, pp. 142–158.

[GGK$^+$83] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir, *The NYU ultracomputer: Designing a MIMD parallel computer*, IEEE Trans. Comput., 32 (1983), pp. 175–189.

[Gou87] M. G. Gouda, *The stabilizing philosopher: Asymmetry by memory and by action*, Technical report CS-TR-87-12, Department of Computer Sciences, University of Texas at Austin, Austin, TX, 1987.

[Had87] V. Hadzilacos, *A knowledge theoretic analysis of atomic commitment protocols*, in Proc. 6th ACM Symposium on Principles of Database Systems, Association for Computing Machinery, New York, 1987, pp. 129–134.

[Hal] J. Y. Halpern, personal communication, 1990.

[Hal86] ———, *Reasoning about knowledge: An overview*, in Theoretical Aspects of Reasoning about Knowledge: Proc. 1986 Conference, Morgan Kaufmann, San Mateo, CA, 1986, pp. 1–17.

[Hem89] D. Hemmendinger, *Initializing memory shared by several processors*, Internat. J. Parallel Programming, 18 (1989), pp. 241–253.

[Her91] M. Herlihy, *Wait-free synchronization*, ACM Trans. Programming Lang. Systems, 11 (1991), pp. 124–149.

[HF89] J. Y. Halpern and R. Fagin, *Modelling knowledge and action in distributed systems*, Distrib. Comput., 3 (1989), pp. 159–177.

[HM90] J. Y. Halpern and Y. Moses, *Knowledge and common knowledge in a distributed environment*, J. Assoc. Comput. Mach., 37 (1990), pp. 549–587.

[HS80] D. S. Hirschberg and J. B. Sinclair, *Decentralized extrema-finding in circular configuration of processes*, Comm. Assoc. Comput. Mach., 23 (1980), pp. 627–628.

[HZ87] J. Y. Halpern and L. D. Zuck, *A little knowledge goes a long way: Simple knowledge-based derivations and correctness proofs for a family of protocols*, in Proc. 6th ACM Symposium on Principles of Distributed Computing, Association for Computing Machinery, New York, 1987, pp. 269–280.

[KK90] S. Katz and K. J. Perry, *Self-stabilizing extensions for message-passing systems*, in Proc. 9th ACM Symposium on Principles of Distributed Computing, Association for Computing Machinery, New York, 1990, pp. 91–102.

[KKM] E. Korach, S. Kutten, and S. Moran, *A modular technique for the design of efficient leader finding algorithms*, ACM Trans. Programming Lang. Systems, 12 (1990), pp. 84–101.

[KMZ84] E. Korach, S. Moran, and S. Zaks, *Tight lower and upper bounds for some distributed algorithms for a complete network of processors*, in Proc. 3rd ACM Symposium on Principles of Distributed Computing, 1984, pp. 199–207.

[Kru79] H. S. M. Kruijer, *Self-stabilization (in spite of distributed control) in tree-structured systems*, Inform. Process. Lett., 2 (1979), pp. 91–95.

[KT86] S. Katz and G. Taubenfeld, *What processes know: Definitions and proof methods*, in Proc. 5th ACM Symposium on Principles of Distributed Computing, Association for Computing Machinery, New York, 1986, pp. 249–262.

[LA87] C. M. Loui and H. Abu-Amara, *Memory requirements for agreement among unreliable asynchronous processes*, Adv. Comput. Res., 4 (1987), pp. 163–183.

[Lam86] L. Lamport, *The mutual exclusion problem: Statement and solutions*, J. Assoc. Comput. Mach., 33 (1986), pp. 327–348.

[Leh84] D. Lehmann, *Knowledge, common knowledge and related puzzles*, in Proc. 3rd ACM Symposium on Principles of Distributed Computing, Association for Computing Machinery, New York, 1984, pp. 62–67.

[LF83] N. A. Lynch and M. J. Fischer, *A technique for decomposing algorithms which use a single shared variable*, J. Comput. System Sci., 27 (1983), pp. 350–377.

[LP90] R. L. Lipton and A. Park, *The processor identity problem*, Inform. Process. Lett., 36 (1990), pp. 91–94.

[Maz89] M. S. Mazer, *A knowledge-theoretic account of negotiated commitment*, Ph.D. thesis, Technical report CSRI-237, Computer Systems Research Institute, University of Toronto, Toronto, 1989.

[Mic89] R. Michel, *A categorical approach to distributed systems expressibility and knowledge*, in Proc. 8th ACM Symposium on Principles of Distributed Computing, Association for Computing Machinery, New York, 1989, pp. 129–143.

[Mis91] J. Misra, *Phase synchronization*, Inform. Process. Lett., 38 (1991), pp. 101–105.

[MT88] Y. Moses and M. R. Tuttle, *Programming simultaneous actions using common* *knowledge*, Algorithmica, 3 (1988), pp. 121–169.

[MT91] M. Merritt and G. Taubenfeld, *Knowledge in shared memory systems*, in Proc. 10th ACM Symposium on Principles of Distributed Computing, Association for Computing Machinery, New York, 1991, pp. 189–200.

[Pea85] G. H. Pfister, W. Brantley, D. George, S. Harvey, W. Kleinfelder, K. McAuliffe, E. Melton, V. Norton, and J. Weiss, *The IBM research parallel processor prototype* (RP3): *Introduction and architecture*, in Proc. 1985 International Parallel Processing Conference, IEEE Computer Society Press, Los Alamitos, CA, 1985.

[Pet82] G. L. Peterson, *An $O(n \log n)$ unidirectional algorithm for the circular extrema problem*, ACM Trans. Programming Lang. Systems, 4 (1982), pp. 758–762.

[PKR84] J. Pachl, E. Korach, and D. Rotem, *Lower bounds for distributed maximum-finding algorithms*, J. Assoc. Comput. Mach., 31 (1984), pp. 905–918.

[PR85] R. Parikh and R. Ramanujam, *Distributed processes and the logic of knowledge*, in Proc. Workshop on Logic of Programs, R. Parikh, ed., Lecture Notes in Comput. Sci. 193, Springer-Verlag, Berlin, New York, Heidelberg, 1985, pp. 256–268.

[PSL80] M. Pease, R. Shostak, and L. Lamport, *Reaching agreement in the presence of faults*, J. Assoc. Comput. Mach., 27 (1980), pp. 228–234.

[RBJ88] A. G. Ranade, S. N. Bhatt, and S. L. Johnsson, *The fluent abstract machine*, Technical report YALEU/DCS/TR-573, Department of Computer Science, Yale University, New Haven, CT, 1988.

[Tau89a] G. Taubenfeld, *Leader election in the presence of $n-1$ initial failures*, Inform. Process. Lett., 33 (1989), pp. 25–28.

[Tau89b] ———, *Self-stabilizing Petri nets*, Technical report YALEU/DCS/TR-707, Department of Computer Science, Yale University, New Haven, CT, 1989.

[Tau91] ———, *On the nonexistence of resilient consensus protocols*, Inform. Process. Lett., 37 (1991), pp. 285–289.

[TKM89a] G. Taubenfeld, S. Katz, and S. Moran, *Impossibility results in the presence of multiple faulty processes*, in Proc. 9th FCT-TCS Conference, C. E. Veni Madhavan, ed., Lecture Notes in Comput. Sci. 405, Springer-Verlag, Berlin, New York, Heidelberg, 1989, pp. 109–120.

[TKM89b] ———, *Initial failures in distributed computations*, Internat. J. Parallel Programming, 18 (1989), pp. 255–276.

[TM89] G. Taubenfeld and S. Moran, *Possibility and impossibility results in a shared memory environment*, in Proc. 3rd International Workshop on Distributed Algorithms, J. C. Bermond and M. Raynal, eds., Lecture Notes in Comput. Sci. 392, Springer-Verlag, Berlin, New York, Heidelberg, 1989, pp. 254–267.

[Tut90] M. Tuttle, *Knowledge and distributed computation*, Ph.D. thesis, Technical report MIT/LCS/TR-477, Department of Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 1990.

[Va95] J. Valois, *A 3-valued wakeup protocol*, Inform. Process. Lett., 55 (1996), pp. 89–93.

[VA86] P. M. B. Vitanyi and B. Awerbuch, *Atomic shared register access by asynchronous hardware*, in Proc. 27th IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1986, pp. 223–243; *Erratum*, in Proc. 28th IEEE Symposium on Foundations of Computer Science, IEEE Computer Science Press, Los Alamitos, CA, 1987, p. 487.

SIAM J. COMPUT.
Vol. 25, No. 6, p. 1358, December 1996

ERRATUM: FAST PARALLEL COMPUTATION OF THE POLYNOMIAL REMAINDER SEQUENCE VIA BEZOUT AND HANKEL MATRICES*

DARIO BINI[†] AND LUCA GEMIGNANI[‡]

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

AMS subject classifications. 68Q25, 65Y05

Remark 2.1, as stated in [1], does not allow one to recover the coefficients of the polynomials $u(x)$ and $v(x)$ which define the Bezoutian $B(u,v)$. In order to overcome this problem, Remark 2.1 should be modified in the following way.

Remark 2.1. Observe that $B(u,v){=}B(u+\alpha v, v)$ for any constant α. Moreover, the last row of $B(u,v)$, i.e., $u_n[v_0,\dots,v_{n-1}]$, provides the coefficients of the polynomial $v(x)$, up to the multiplicative factor u_n, as well as its degree m. Now we assume for simplicity that $u_n = 1$ and show how we may recover the coefficients $\tilde{u}_i$, $0 \le i \le n$, of any polynomial $\tilde{u}(x) = u(x) + \alpha v(x)$ satisfying $B(\tilde{u}, v) = B(u,v)$. First, observe that, since α is an arbitrary constant, we may choose $\tilde{u}_m$ arbitrarily. In the case where $m = 0$, the first row of $B(u,v)$ immediately yields the coefficients $\tilde{u}_i$, $i = 1,\dots,n$. Otherwise, if $m > 0$, then we consider the vector $\mathbf{b}^T = \mathbf{e}^{(m-1)T}B(u,v)$ and (only in the case $m < n-1$) the vector $\mathbf{c}^T = \mathbf{e}^{(m)T}B(u,v)$. Here $\mathbf{e}^{(i)}$, $0 \le i \le n-1$, denotes the $(i+1)$st column of the $n \times n$ identity matrix. From the last $n-m$ equations of $\mathbf{c}^T = \mathbf{e}^{(m)T}B(u,v)$, that is,

$$[c_{m+1},\dots,c_n] = [u_{m+1},\dots,u_n]\begin{pmatrix} v_m & & O \\ \vdots & \ddots & \\ v_{2m-n+1} & \cdots & v_m \end{pmatrix},$$

where we assume $v_i = 0$ for $i < 0$, we obtain $u_{m+1},\dots,u_{n-1}$ by solving an $(n-m-1) \times (n-m-1)$ triangular Toeplitz system. From the first m equations of $\mathbf{b}^T = \mathbf{e}^{(m-1)^T}B(u,v)$, i.e.,

$$v_m[u_0,\dots,u_{m-1}] = -[b_0,\dots,b_{m-1}] + [u_m,\dots,u_{2m-1}]\begin{pmatrix} v_0 & \cdots & v_{m-1} \\ & \ddots & \vdots \\ O & & v_0 \end{pmatrix},$$

where we assume $u_i = 0$ for $i > n$, we may recover $u_0,\dots,u_{m-1}$ as functions of the parameter u_m. Now, by substituting an arbitrary value $\tilde{u}_m$ for u_m, we determine the values $\tilde{u}_i$ of u_i, $i = 0,\dots,m-1$.

REFERENCES

[1] D. BINI AND L. GEMIGNANI, *Fast parallel computation of the polynomial remainder sequence via Bezout and Hankel matrices*, SIAM J. Comput., 24 (1995), pp. 63–77.

*Received by the editors February 14, 1996; accepted for publication February 15, 1996.

[†] Dipartimento di Matematica, Università di Pisa, 56127 Pisa, Italy (bini@dm.unipi.it).

[‡] Dipartimento di Informatica, Università di Pisa, 56127 Pisa, Italy (gemi@di.unipi.it).